UiO : Department of Informatics
University of Oslo

# Program and diagram comprehension of first year students

Line Moseng

Master's Thesis Spring 2015

# Abstract

How to improve the learning methods for beginners and students in the earlier stages of learning programming is a matter debated among students, lecturers and developers. This master thesis investigates whether visualisation of programs using class and object diagrams when programming seem to improve the comprehension of first year programming students and whether there is a connection between the quality of their program and diagrams.

A quantitative analyse is used to find the connection between 56 sets of programs and diagrams created by first year programming students at a course in object-oriented programming at the University of Oslo. The programs and diagrams were created for a delivery to a mandatory assignment in the course. We have set up some criteria and weighted them with points that we have used to evaluate both programs and diagrams. The total scores of a delivery is used to categorise the deliveries for then comparing the best and poorest programs and diagrams.

We found students struggling with null-pointers and method lengths in their program code. Studying the diagrams, we saw that students from all parts of the point scale struggled with illustrating abstract classes in the class diagram and in the object diagrams that few managed to mark next-pointers in the list objects.

Our research have not shown a connection between program and diagram quality and do not suggest that creating diagrams improved the comprehension of programming for the students taking this course. This may be caused by the lack of focus on how and why to create diagrams in the course curriculum and the fact that the program the students had to create were quite big. What we found was that of the students delivering poorly written code, there were both good and poor diagrams. The same we found for those that delivered good code.

# Acknowledgements

I want to thank my supervisors Ragnhild Kobro Runde and Jens Kaasbøll for guidance and feedback. I also want to thank Geir Kjetil Sandve for reviewing my statistical parts, Roger Antonsen for inspiring conversations and Marte Svalastoga and Sindre Wetjen for taking their time to read my thesis while it still was a mess.

I want to thank my family for all support, my dear friends for making me laugh when I thought I couldn't, my fellow students at the study room for making the long days seem shorter and the guys at the student bar for serving me good coffee in the morning and necessary beer in the weekend.

Finally I want to direct a huge thanks to all the amazing students at the Department of Informatics for creating an unique studying and social environment and for making my student years fantastic.

**I've had the time of my life.**

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Teaching assistants and lecturers in programming courses for first year programming students often experience that their students start solving the problems at hand straight away, without thinking through solution strategies first (Rist, 2004). If they get stuck at some point while implementing their solution, they rarely back up to consider whether they have misinterpreted the problem, selected a solution that will not solve the problem or if it is a poor solution. Instead they can get stuck in a loop between trying to find errors in their implementation and checking whether the problem is solved. To improve the way we teach programming to first year students, we also have to know how these students approach programming tasks. Visualisation and abstraction are often seen in lectures from many types of computer science courses as a technique to improve the comprehension of concepts.

A programming introduction course at Harvard College for both computer science majors and non-majors aims at teaching abstract thinking, data structures, algorithms, databases and software development, using multiple programming languages including C and the "drag-and-drop" language Scratch (Harvard College, 2015). The course is focused on teaching how to think algorithms and problem solving and they are showing great results (Malan, 2010).

The US Berkeley EECS Department course CS10 - The Beauty and Joy of Computing is teaching programming and complex concepts by programming in the Scratch developed language Snap! and using every day images when explaining curriculum (UC Berkeley EECS Department, 2015). The course is intended for non-majors in computer science but still teaches advanced concepts like lists, recursion and concurrency, focusing on the comprehension of the concepts rather than being able to understand language dependent functionality.

We want to study the students comprehension of programs they write themselves compared to diagrams they create to illustrate the program. We will take a look at how they manage references and implements data structures in their programs. This is central concepts in many computer science courses, so hopefully our results can apply to other universities as well.

In this thesis we will focus on programming in Java when it comes to language specific problems and the assignments given to students. This is mostly because we want to analyse student deliveries from the University of Oslo's course in object oriented programming that is using Java. This course is mainly taken by students in their second semester and is a continuation of the beginners courses at the university.

To research comprehension we will look at some specific problem areas that might show what poorer students are struggeling with and that students with seemingly good comprehension manages. For some of the areas we can check whether the students has been implemented it or not, while others requires an evaluation of the way it is implemented.

## 1.1 The author's personal motivation

The motivation behind the choice of research comes from my own experience as a teaching assistant at both a beginners course in Java and the second semester course in object oriented programming we are looking into in this thesis. I have seen students struggling with programming larger mandatory assignments while they seemingly do not take their time comprehending and analysing the actual problem description. I wanted to study possible reasons and solutions for this and I hope that my research can in some way help new students in learning programming more easily.

## 1.2 Motivation and background

Creating diagrams and visualising programs as a way of planning the implementation is commonly used by software developers. We want to find out whether illustrating, in addition to coding, the programs will help the students in becoming better programmers.

### 1.2.1 About object-oriented programming

Like several courses in object oriented programming at other universities around the world, the first year programming course we are examining teaches the object-oriented paradigm using Java. Java is a multi-paradigm language and the paradigm most used and often associated with it is object orientation. This paradigm is based on the concept of the programs being structured into logical objects, which are data structures containing data, and the objects' attributes and methods.

## 1.3 Research questions

In this thesis we hope to find relations between first year computer science students comprehension of programs compared to diagrams describing the program. The questions we want to answer is:

- Will visualising the program ahead of programming it help the students in writing better code and to comprehend the concepts at hand?
- Is there a connection between good code and good program diagrams?

These questions will be further elaborated in 3.2.4.

## 1.4 Overview of the thesis

In this thesis we will look into some of the concepts we have experienced that first year computer science students struggle with comprehending, and choose the ones we find interesting to look further into. For a concrete example of how programming is presented to students, we will take a look at a specific introduction course in object oriented programming. We analyse a set of programs written by students taking this course in order to answer our research questions.

In **Chapter 2** we will present the problem areas we find that programmers in the early learning stages struggle with and take a look at the university course we are researching.

In **Chapter 3** we will explain our approach at finding answers to our research questions from the previous section, elaborate them and what we are trying to achieve. We will describe the student deliveries we are taking a close look, the methodology, analyse and evaluation criteria used in our research.

In **Chapter 4** we present the results from our analysis and their significance.

**Chapter 5** discusses our results and whether they can apply to more situations than the examined course.

**Chapter 6** concludes our work, presents some suggestions for improvement for the course and some thoughts for future work. The research questions from section 1.3 and 3.2.4 will also be answered.

# Chapter 2

# Problem Areas

In this chapter we will introduce some programming concepts we find, based on literature and our own experiences as teachers, first year students struggling with on their road to becoming better programmers. We will also present the course we are collecting data from for our research. Finally we will recap some related studies.

## 2.1 Planning the data structure

When given a problem to solve or a program to implement, experienced programmers plan how the implementation should be before they write the program. They have learned techniques and software development methodologies that makes the development process more efficient. To plan what data structures to use and how to divide the problem into smaller components like classes and methods is becoming more important when the programs get bigger and more complex. While students often get introduced to some strategies for doing this, we have witnessed that they rarely test them, resulting in most students skipping the whole pre-programming phase and jumping straight on to programming, without much of a plan or strategy.

We get the impression that most first year programming students don't know how to structure their program, they don't think about what to do before they start writing code. Designing a system is more difficult than implementing concrete parts of it and requires experience (Börstler et al., 2008). In lack of experience there is a need for strategies to design software systems.

### 2.1.1 Diagrams as a development technique

Creating different types of diagrams that illustrate the program are common tools in many software development techniques (Miles and Hamilton, 2006). Diagrams can be used for documentation of a program as well as for planning and developing.

There are several different types of diagrams displaying different aspects of a program and they are usually classified as either *behaviour*

*diagrams*, its sub-category *interaction diagrams*, or *structure diagrams* (Pilone and Pitman, 2005). Behaviour diagrams describes the behavioural features of the program, interaction diagrams illustrates the data flow while the structure diagrams explains what has to be present in the program. We will look into two types of structure diagrams that are relevant for the university course, the *class diagram* and the *object diagram*.

#### Class diagrams

Class diagrams illustrate the relationships between the classes in a program, what super and sub classes and what interfaces are implemented by which classes. The class diagram do not need to include much more than the class name, arrows to its super class and arrows to the implemented interfaces, if the classes implements or extends something. Interfaces are often marked with having the interface name in italic font.

#### Object diagrams

Object diagrams illustrate what objects that is created in a program and how they are connected. References in one object to another is therefore included, often drawn as arrows. One of the goals by creating object diagrams it to make it easier to write the equivalent code by following the references between the objects (Lingjærde et al., 2011).

While creating class diagrams can be done by illustrating the class signatures found in the code, object diagrams is a more abstract concept, as they illustrate parts of the program execution.

## 2.2 Java specific problem areas

As there are many problem areas that are dependent on the programming language, we will focus on those related to object orientation in Java programming. Java, designed by James Gosling and Sun Microsystems and today developed and maintained by Oracle Corporation, is a widely used, multi-platform programming language.

### 2.2.1 Classes, inheritance and abstract classes

Programs written in Java consist of classes. A class can extend another, making it a sub class, it then will inherit the fields and methods that is not declared private of the class it is extending, its super class. This is used for eliminating the need to write duplicate code when creating different classes that have some equal functionality.

A class can also be declared as abstract, so that one can not create objects of it, requiring other classes to extend it to use its functionality.

We can also write interfaces, which is an specification of what a class have to contain. Interfaces do not implement any functionality, only specifying methods and fields that must be present in any classes using

this interface. A class implementing an interface must implement what the interface has specified.

### 2.2.2 References

References are an essential part of programming in Java. In order to access an object, one need a reference to it, since the data structures in Java is based on objects and the references are connecting them. This makes the understanding of what a reference is a basic skill to learn.

We can see that many students seemingly lack comprehension of the difference between object comparison and reference comparison, e.g. "does these two objects contain the same" versus "does these two references point at the same object". This can be a result of students not understanding the difference between those two, meaning that they have missed an essential part of the programming language.

### 2.2.3 References versus primitives

All variables in Java is either a *reference* or a *primitive*. Examples of primitives are `int`, `boolean`, `float` and `char` while references can be of any type we define, e.g. `Car`, `Person`, `Book`, etc. or built in types like `HashMap` or `Character`. Primitives has a pre-defined size, so when declaring a primitive the computer sets aside a location in memory at this exact size. The variable then refers to that location. When declaring a reference variable, like to a Car object, Java does not know how big this car is, so the variable contains just enough memory space to hold a reference. If we also initialise the variable, the reference variable will change to point at the actual location of the object.

In Java, the String class is implemented in the language to look like a primitive even though it's not. String can be initialised in the same way as a primitive type and it is constant, meaning they cannot be changed after creation. If a change is made to the string, Java creates a new String object in stead of altering the old. To create an instance of a class, one have to use the keyword `new` in front of the class name and call the class constructor. Some students might find this confusing, making it more difficult to understand the difference between references and primitives.

### 2.2.4 Null pointers

As in many programming paradigms, it is important to understand what a reference to null is, when it occurs and how to avoid program crash because of it. In Java, trying to access an objects attributes without checking if the object reference points at null can cause the program to crash with a `NullPointerException`. Null pointers can occur of many reasons; from input streams like keyboard or files and from data containers not containing data. When writing complete programs, it is important to take care of null pointers so that the program does not crash.

Even though null-pointers is a discussed subject and according to clean code principles one should not have to check for it (Robert C. Martin, 2008), it is necessary to comprehend the basics, like null pointers, when learning programming.

### 2.2.5 Data structures

As a continuation of references, and as a way of testing how students comprehend them, we will discuss the subject of data structures as well. Data structures describes ways to organize data for storage in a computer. When implementing complex data structures one has to deal with references to other objects and in order to make the implementation good it is important to use the references correctly.

#### Linked lists

Linked lists is a commonly used and one of the simplest data structures. A linked list often has a reference to the first element in the list, the list head, and normally also to the last element, the list tail. The list normally has a node class, which is the type of the head and tail references, that has two essential purposes: To keep track of an element inserted in the list and to link to the next node in the list. There are several types of linked lists. They can have references to both next and previous node, making it a *doubly linked list*. If the Nodes only have reference to the next, it is a *singly linked list*. The list can be *circular*, meaning that the last node in the list has the first node in the list as its next. With circular lists it is important to know when the program is looking at a head/tail node and when it is a random node in the list, if the program does not know, it can end up checking all next node for ever, making an eternal loop. The list can also be *linear*, in which case the last node in the list has a next pointer to `null`.

### 2.2.6 Code principles

There are several code principles, norms and guidelines of how to write programs that does not only work, but also is well written. Most programming languages have their own conventions on how the code should look as well. Oracle Corporation do not maintain the official Java conventions, which were last updated in 1999.

An example of a language independent programming principle for object-oriented programming is *clean code*, which describes how to divide code into classes and methods, explains good and bad code practices and how to write readable and easily understandable code (Robert C. Martin, 2008). The most common code principles has to do with readable and easily maintainable code. As C++ programmer John F. Woods explained it:

> "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability." (Woods, 1991)

SOLID is another principle, much applied to object-oriented programming, which describes, among other things, the single responsibility principle, which states that a method should have *one and only one* responsibility (Robert Cecil Martin, 2003). This is a central idea in the UNIX philosophy also, *"do one thing and do it well."* (Raymond, 2003).

## 2.3 The course

The course we will focus on in this thesis is the second course in Java and programming at the Department of Informatics at University of Oslo (UiO), Object oriented programming (the course code is INF1010). In order to apply to this course one must have passed a programming introduction course. Most of the students have background from Java or Python. The course is neither an introduction course nor a traditional algorithm and data structure course, but a course in between, continuing the basic programming concepts and giving the students more experience and being a lighter introduction to more complex concepts than the beginners courses touch.

The students must have passed one of the two programming beginners courses at the university before taking this course. The course has six mandatory programming exercises that is delivered to a teacher assistant for approval, in which all of them must be approved in order to qualify for the final exam. The students are given two tries at passing an assignment if the first attempt is good enough that it seems the student actually have put down a fair amount of work. The course teaches linear doubly and singly linked lists and uses diagrams in lectures and lecture notes for illustrating the concepts.

### 2.3.1 Learning outcome

The following learning outcomes are listed on the course page:

"After completing this course you will have a thorough knowledge about, and be able to use yourself when you program:

- subclasses, abstract classes, interfaces, virtual methods, abstract data types and alternative implementations
- cooperation between objects, including programming with server-clients and peer-to-peer programming.
- some important data structures such as one way and doubly linked lists and binary trees with associated algorithms

After completing this course you will have a good knowledge about, and be able to use yourself when you program, simpler versions of:

- recursion
- graphical user interfaces with event programming

- parallel programming, shared data, synchronisation and threads
- the Java class library ” (Department of Informatics, 2015)

We emphasize that the students should, after completing the course, have understanding about subclasses, abstract classes, linked lists and cooperation between objects. We therefore think this course is suitable for our research, introducing concepts complex enough that illustrating the programs can be useful for easier comprehension of the data structures. Even though the course aims at learning some widely used programming concepts, the course is anchored in Java, making it essential to master the language as well as the concepts. The course is using standard lectures, lab sessions and classroom teaching with teaching assistants, plenary live coding sessions and some interactive activities such as making students pretending to be objects in linked lists and performing sorting algorithms.

### 2.3.2 The diagrams

Both the lecture and the curriculum notes uses diagrams to illustrate the concepts taught (both class and object diagrams), but they never provide an overall definition on how the students should draw these diagrams and states that the diagrams do not follow Unified Modelling Language (UML) diagram definitions even though they look a bit like them (Gjessing, 2012). The first diagrams illustrated are class diagrams containing detailed information of the class, the methods even contain program code. At the same time these diagrams are introduce, the lecture notes states that the diagrams should be detailed in the beginning but as the course goes on, they can be simplified. In figure 2.1 an example of an object diagram from the lecture notes is shown, both containing two class fields, pointer to an array and program code in the main-method. The diagram from the lecture notes stores integers in the String objects, which could be confusing for the students. This is perhaps not a very good example to introduce diagrams.

Another interesting observation in the lecture note diagram in figure 2.1 is that the main class is framed by a dotted, black line while the String objects are framed by a solid blue line. The reason for these different frames could be that the main class is not an object, while the String object is, but as this is not explained in the lecture note, this is our guess and we do not expect the students to know the reason.

We also note that we have not found an explanation for what an arrow illustrates. While the lecturers might think it is implicit, the course requires no previous knowledge of diagrams, making it a need for explaining what an arrow represents.

A second problem is that the students are not taught how to create diagrams, they just see a lot of examples and get a note that explains *some* of the features like how to mark private and public attributes and methods. There is a lecture note on linked lists in general, that also has some explanations of how to draw lists, but it is not defining how to create diagrams in general, just explaining how to create a diagram of the shown code example (Storleer, 2013). The curriculum book teaches UML

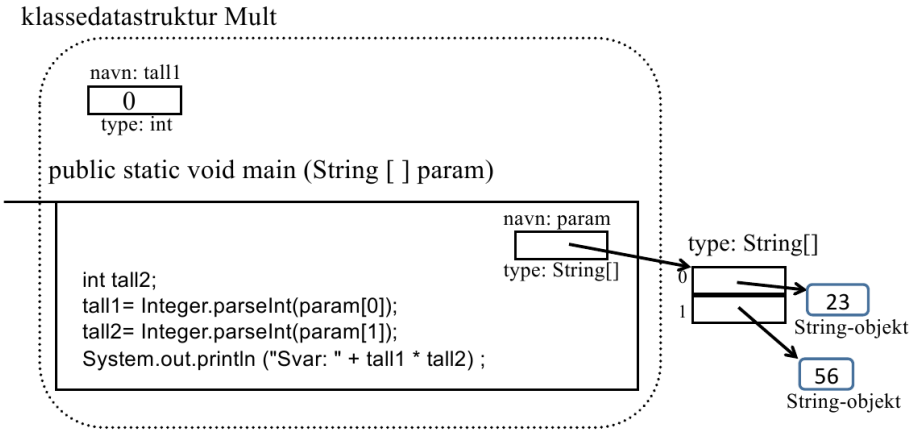


Figure 2.1: Class diagram example from the first lecture note.

diagrams and even though this chapter is only cursory there is a possibility that some students read this for guidance. For comparison figure 2.2 shows the first object diagram from the curriculum book. The most notable difference is the fact that there is no actual program code in this diagram, nor in any other of the diagrams in the book.

Neither have we found any explanations given in the course material of how to make use of diagrams when planning and writing a new program. Still, creating diagrams is often included as a part of the final exam.

## 2.4 Related research

In this section we will briefly go through some of research related to the topic of diagrams and visualisation combined with learning and teaching.

### 2.4.1 Mental and physical images as an learning technique

The use of hooks and props, which is mental or physical images, to complement lectures and programming exercises are discussed by Owen Astrachan. To use drawings or props like building blocks for children in addition to verbal explanation and viewing code can help the students create a mental model of data structures concepts (Astrachan, 1998).

To visualise programming concepts, illustrating data structure and abstracting away the programming language can be used for explaining and understanding programming, like in computer science professor Paul Curzon's Computing Without Computers (Curzon, 2002).

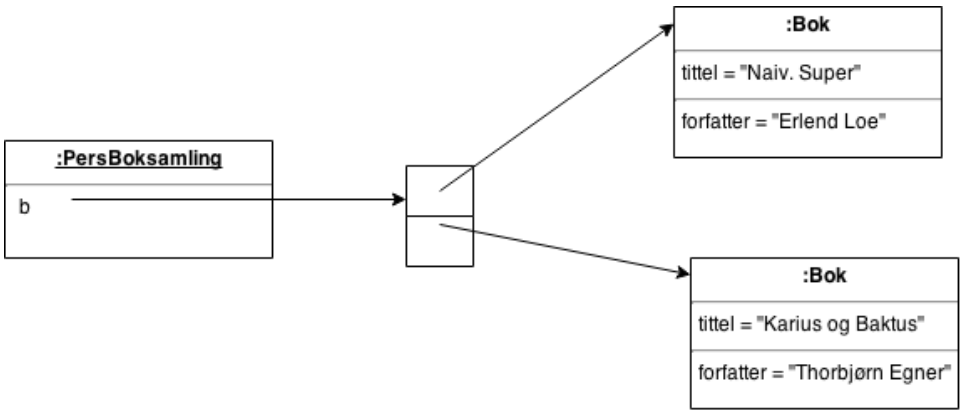


Figure 2.2: UML object diagram from curriculum book

### 2.4.2 Creating diagrams as an comprehension technique

Creating diagrams of different sorts are a common way to plan the structures, flow and architecture of a program. The diagrams are a common part of many other development techniques, whether the diagrams are standardised for communication of the program, meant to structure the developers thoughts or a step in the process of comprehending solutions for a problem.

Research from Western Carolina University show that there is a correlation between students creating memory diagrams of simple programming concepts and their understanding of the corresponding code fragment (Holliday and Luginbuhl, 2004). Memory diagrams is quite alike the object diagrams in the course we are researching, meant to illustrate a piece of code at a certain time in the program run.

Results from a research at the University of Wales, by Thomas, Ratcliffe and Thomasson, suggested, on the other hand, that drawing object diagrams in order to understand written code did not appear to help the studens' comprehension. The setting was multiple-choice questions, where they used the test results to compare score, diagrams and code (Thomas, Ratcliffe and Thomasson, 2004).

Other fields than computer science have also benefit from using diagrams for increasing understanding. Experiments and research show that illustration models have helped students understanding how radars and electrical components work as well (Mayer, 1989).

### 2.4.3 Aspects of problem solving

When writing programs, one basically are solving problems. In a problem solving process the first step is to interpret the problem, then to think of

solution strategies, before implementing the solution. Schoenfeld found that when less experienced mathematics students were to solve a non-standard problems they would read the problem description and quickly chose a solution strategy before trying to solve the problem using that approach. If the approach did not solve the problem the students continued working on it despite the fact that they did not make progress. They could neither explain *how* the approach would have solved the problem. For comparison, the experienced mathematician would spend time analysing the problem, "trying to make sense of it", which the students never did. The experienced mathematician did not commit to a solution, but stopped implementing if it seemed to be a dead end and instead go back to analysing (Schoenfeld, 1992).

Problem solving in programming can also be done by following the steps: Reading the problem description, analysing and understanding it, exploring and planning the implementation of the solution, then implementing it, and last verifying that the implementation actually did solve the question. There can be difficult for lesser experienced programmers to separate the concepts to implement from the programming language. This lack of abstract thinking can make the analysing and planning steps in the problem solving process harder if the students are thinking about both what to implement it and how to do it at the same time. We want to see whether using diagrams to analyse a program description and using them to plan the implementation of the solution seem to make the programs better by comparing diagram quality with code quality.

# Chapter 3

# Approach

In this chapter we will describe our approach to answer our research questions. We will describe what we chose to analyse and how we planned to do it before we will go into the details of our system and criteria used for analysing the student written programs in detail. We want to study whether there is a connection between the quality of the code and the quality of diagrams describing the program. Our approach should give us some indications about the students comprehensions and misunderstandings of their programming, by comparing the strengths and weaknesses of the diagrams and code.

The areas we want to use for evaluation was introduced in chapter 2 and we will go into details for this evaluation in section 3.5.

## 3.1 Attempted approaches

First we will start by explaining some initial studies that led us to the approach in this thesis, and what we learned from these.

### 3.1.1 Alternative teaching methods

As a first approach for learning about the students comprehensions and misunderstandings, we wanted to teach some students the concepts of creating object diagrams before writing code for structuring thoughts and as a process of develop solution strategies. We started off by teaching the students in one classroom in the beginning of class and the rest of the time observing how they approached the programming exercises and whether they would create diagrams them selves. The students were encouraged to create diagrams of the assignment implementation before programming.

We observed that in the classroom where the students got teaching, some of the students from the research group created diagrams and also let us study them after class. The observations showed that the variation in the diagram quality were widely spread, from “correct” to “poor”. We also observed students from another classroom that did not receive this teaching, only the standard lectures in the course, where none of them

created diagrams, even though this is encouraged by the lecturers of the course.

We did not pursue this approach because the lecturers at the course thought it would steal valuable time from the students that they could have used for programming the mandatory assignments. Thus, we never got to study and compare the two groups in detail for whether their program code differed notably.

### 3.1.2 Interviews

As a second approach, we interviewed some volunteer students from the classroom we taught, as they explained the object diagram they had created in class. Then we asked them to drawing a new object diagram of a program they had just finished programming, a network between friends and enemies. From this we learned that students having quite good comprehension of programming can create diagrams with quite different quality. One drew diagrams like the ones in the lecture notes, the other were quite sloppy when drawing and the third did not represent pointers in a logical way. Still, all three explained their program quite well and what seemed to be a good implementation of the problem while they were drawing the diagrams.

With only three students volunteering to participate, we did not get more than three volunteers, whom all seemed to have quite well comprehension of their written program. We did not have much basis to go on for our research and we decided to change approach to one that did not depend on us having students willing to help us

## 3.2 The approach in this thesis

For our chosen approach we will analyse a large number of programs and diagrams created by students at the UiO course in object-oriented programming. From the position as a teaching assistant, the author have observed students working on mandatory and weekly assignments given in the course. We chose to analyse deliveries from a specific assignment given in the middle of the semester. These deliveries should consist of a program written in Java, a class hierarchy diagram and an object diagram describing the container data structures. See section 3.3 for description of the assignment. We want to compare the program code with the class diagram and the object diagram in order to find out if they understand what they are doing.

### 3.2.1 Analysing deliveries

We will divide the deliveries in three categories according to their scores. The three categories will not be of exactly equal size, as we will look at how the deliveries are distributed along the point scale and use this to find places where there seem to be “gaps”, distinctions, that creates a natural division.

Still, all three categories should contain many enough student groups for them to be statistically comparable. We will refer to the three categories as *poor*, *middle* and *best*.

### 3.2.2 The analysis

When analysing the data set, we use a list of criteria for giving points to the deliveries and finding errors, which is described in section 3.5. We distinguish between errors and failing to fulfil a criteria. When a delivery fails a criteria, we write down what they did instead, sometimes this can be an error, other times an alternative solution. Thus, failing to fulfil a criteria can show errors in a delivery but also alternative thinking and problem solving. A criteria can point out multiple errors, and a type of error can be found in multiple criteria (e.g. null pointer errors).

#### The point system

We will weight the criteria with 1 or 2 points, the criteria worth 1 point is considered small tasks, often only consisting of one or two code lines while the criteria worth 2 points requires more work, like implementing a method. When given 0 points, the delivery fail to meet the criteria due to wrong implementation or not it not being implemented at all. If a delivery gets 1 points on a criteria worth 2 points, it means that the criteria is not fulfilled, but the implementation is good enough to get *some* credit for the attempt. In order to get 2 points, the implementation has to be good, but not necessary perfect.

We accept different naming of the classes and methods, as long as they are reasonable named according for their functionality.

### 3.2.3 Criteria for categorising deliveries

Before we can categorise the deliveries, we will use some criteria for evaluating them. Some of the criteria will be specific to the assignment and some more general, reflecting programming norms and practises. We will select the ones we find the most relevant for this master thesis and look through a large number of student deliveries to see whether they fulfil the criteria. The area we find most interesting are based on our experiences with the students at the course (both the same and previous semesters), but this is somewhat subjective and others would perhaps have chosen different area and criteria for inspection.

### 3.2.4 Specific research questions

We will categorise the deliveries in three turns, according to the score of the code criteria, to the score of the diagram criteria and to the of both - the total score on the entire delivery. We will classify typical errors and mistakes that we find in the deliveries in order to find out what is the most common errors and miscomprehensions. By dividing the deliveries into these three

categories we hope to find out what the deliveries in the categories are good at, where they struggle and how far an average poor student lies behind an average good student.

Elaborating the research questions presented in 1.3, the goal of categorising the student deliveries is to answer these questions:

- What do the deliveries have in common? What do the poorer deliveries have in common and what are the similarities between the good deliveries?
- Is there a connection between the program (the actual code) and the diagrams? Are there good code without good diagrams? Is there good diagrams with bad code?
- What is missing to elevate the poorer students up to an average or good level?
- Can we state something about the students understanding and miscomprehension by comparing code and diagrams?

After analysing the deliveries, we will look into what the typical weaknesses and strengths of the poor and the good group is by describing some typical examples of deliveries in the two categories in more detail.

## 3.3 Description of the assignment

There were only two of the six mandatory assignments in the university course that included both creating and delivering diagrams, which were necessary for us to compare diagrams with code. The first assignment was given the second week in the semester and the fourth assignment in the semester. The first was quite small, just creating a system containing persons and link them together as friends of enemies. The object diagram the students should create should only contain one object. Thus we chose to examine deliveries to the fourth assignment since both the code and the two diagrams contained more features and work.

The assignment is to construct a medical data system, containing doctors, patients, medicine and prescriptions. The main programming concepts used is linked lists, iterators, generics, subclasses and nodes. Many students used a programming technique called pair programming for this assignment, and thus solved and delivered in pairs. [1]

The first part of the assignment is to draw the class hierarchy for doctors, patients, prescriptions and medicine described in the assignment text. Figure 3.1 shows a solution sketch for the Medicine with 3 subclasses for the type of drug (strongly narcotic, addictive or normal), which again has 3 subclasses for the form of medicine (pill, liniment or injection). Figure 3.2 shows an alternative solution where Medicine has 9 subclasses.

[1] Pair programming is a software development technique where two developers work together at the same computer. One is having the role as the driver, having control of keyboard and writing the code, while the other is the observer, reading and evaluating the code being written. The roles are switched often.

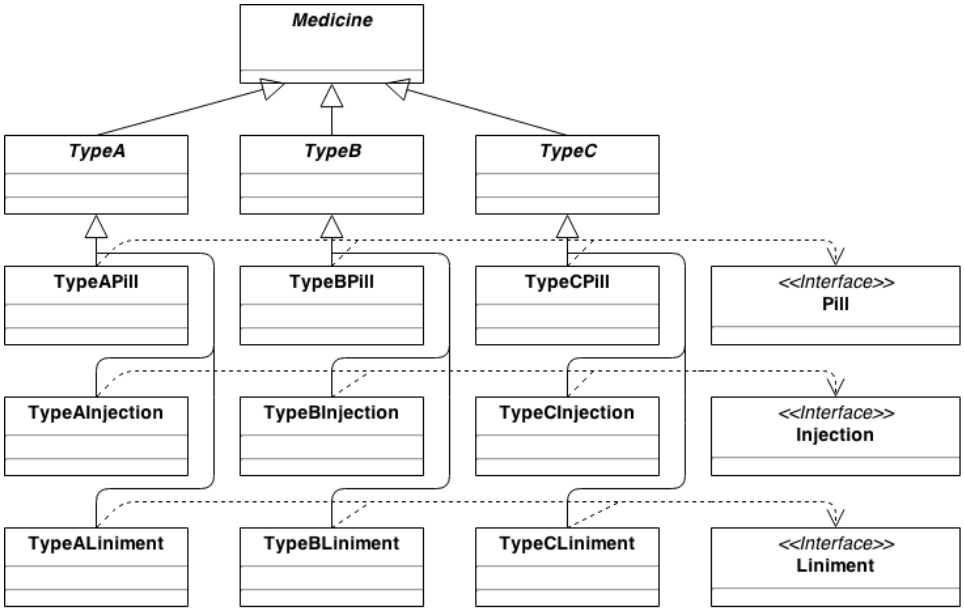


Figure 3.1: Solution sketch for the Medicine class hierarchy with abstract classes for Type

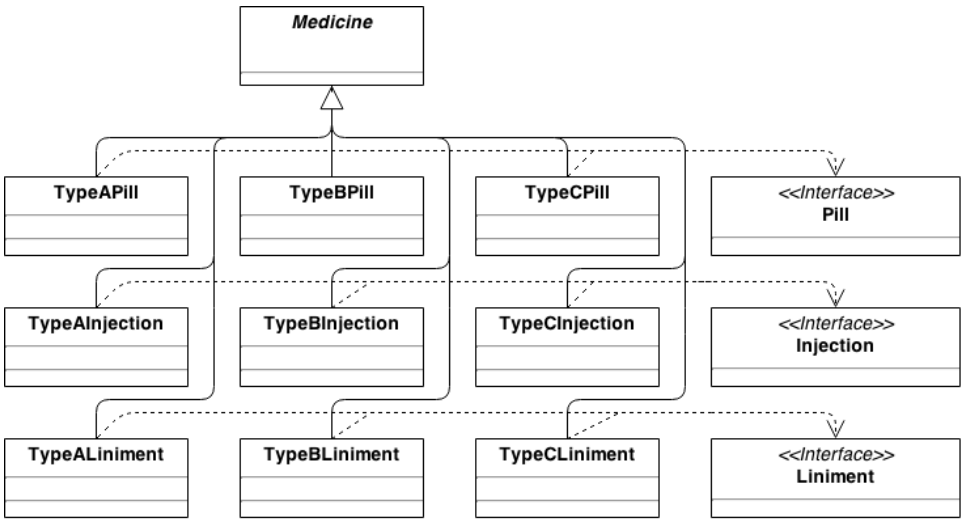


Figure 3.2: Solution sketch for the Medicine class hierarchy with 9 subclasses of Medicine.

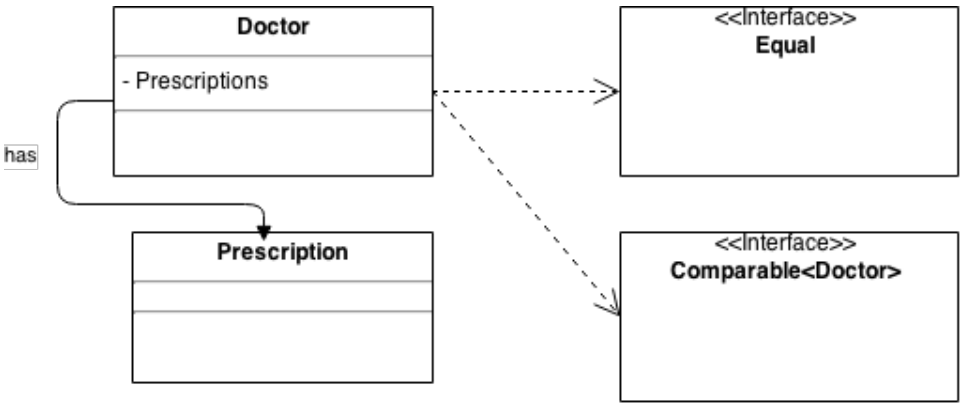


Figure 3.3: The rest of the class hierarchy.

Figure 3.3 illustrates the remaining classes of the class diagram. From the assignment description we interpreted that the class diagram should show that class Doctor has a relation to the class Prescription.

The second part is to implement the classes described in the hierarchy diagram. Next is to draw the *object diagram*, which should illustrate how the containers for the objects from the first part should be implemented. There are mainly three container classes, one uses a generic array, one is a single linked sorted list containing comparable elements and the third is a single linked list of prescriptions. The latter class has two subclasses, one following the "last in, first out" (LIFO) concept and the other "first in, first out" (FIFO). Some objects must also be included. After creating the object diagram, the students must implement interfaces describing the generic containers and then the container classes.

Figure 3.4 shows an example solution of the object diagram. An arrow pointing at a cross is illustrating a null pointer. The diagram does not show the parts of the assignment that we do not include in the analysis in this thesis. Worth noting is that this assignment is not the first time the students are asked to create a diagram of a program they shall create. The first mandatory assignment in the course requires an simple object diagram delivered in addition to the program code. The diagram should illustrate one Person object and its fields for name, two Person references (referring to null) and two Person arrays that is not containing any references. Neither is it the first time the students are creating a linked list.

The last part of the assignment is to write utility tests for the classes (which they did not have to include in the delivery to their teaching assistant) and then write a user interface for registration, deletion and printing patients, doctors and prescriptions. The assignment text (in Norwegian) can be found in Appendix A.

The assignment description is very specific about what should be implemented, even with details about what fields and methods the classes should have. It is a bit like a recipe where the students are left not left with much freedom to make choices or discuss solutions.

The students are allowed to ask the teaching assistants in the course for help, and since there are about fourteen of them, there is a possibility that some students get a lot of assistance, while others get none. This is not possible for us to take into account first of all since our data set is anonymous and because we have not enough knowledge about the students and the teaching assistants to find this out.

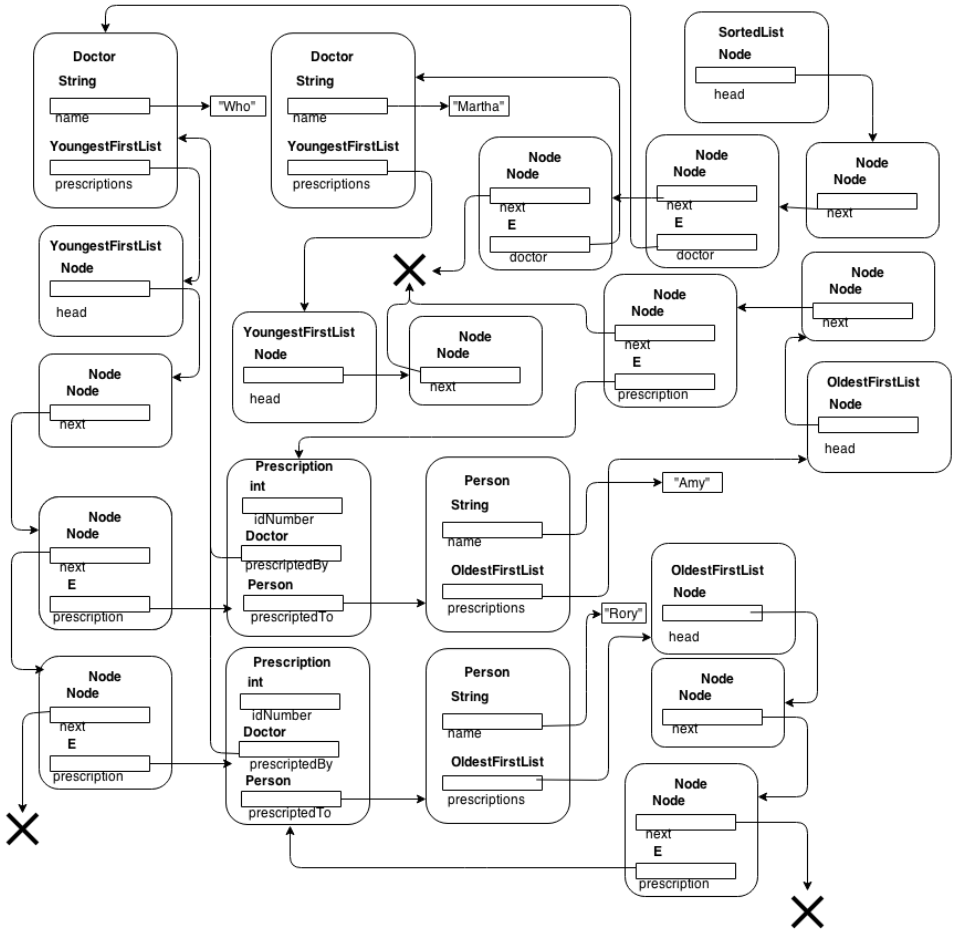


Figure 3.4: Solution sketch for the object diagram. It is only showing the parts relevant for this master thesis.

## 3.4 Methodology

### 3.4.1 The data set

In this thesis we have used a data set containing deliveries from 57 student groups. The groups consisted of one or two students. The analysed deliveries contained both those that were good enough to be approved and those who weren't. If the delivery was not good enough, it was disapproved, and the student group would have to deliver again in order to continue with the course. The student groups can, in other words, make multiple attempts to pass the assignment, but in this thesis we only consider the first attempt. The student groups can upload their delivery as many times as they want before the deadline, we look at the one last uploaded. This is because this is what we assume that the students consider is their best attempt to solve the assignment. The student groups that does not get their delivery approved will receive comments from the teaching assitant about what is wrong and what their assignment is missing. Because they get so specific feedback, we decided to remove this variable by only looking at their first attempt to pass. The students can get feedback from their teaching assistant and other students before the first delivery as well, but this is a variable that is more or less impossible for us to take into account with our approach.

We chose to look through 57 deliveries because it is an amount that is reasonable to work with considering the time aspect of this thesis, but still an amount that we thought would be a representative, random sample.

### 3.4.2 Privacy

In order to handle students deliveries, we had to assure that they were anonymous and handled with care. We used an anonymous data set of student deliveries for analysing, in order to protect the students privacy. The data set was downloaded from the delivery system at the university after the course was completed and the exams had been corrected. Thus, our analysis have no affect whether the students gets the assignment approved, future assignments or their results on the final exam. It is not possible to track the students from our data analysis or results. The deliveries were permanently deleted after the analysis was finished.

### 3.4.3 The tools we have used

To analyse the student deliveries we have used a spreadsheet to write down the scores on all criteria and all the errors we found. The spreadsheet was afterwards imported into Python for calculation of the results and used a package named Matplotlib in order to produce histograms and plots. To display the distribution of the delivery scores, we used histograms and for average scores on main categories we used plots.

### The statistical tools

Since this is partly a quantitative analysis, we use some statistical tools to find significance of our found data. The test is one-tailed, meaning that the chosen alternative hypothesis states that one side is concidered larger thant the other. The null hypothesis on the other hand states that both sides of the distribution is considered equal. An example is:

$$H_0 = n_e = n_n$$
$$H_a = n_e > n_n$$

Where $n_e$ is the number of people from England watching the TV-series Doctor Who and $n_n$ is the number of the people from Norway watching the TV-series.

**Fisher's exact test** In order to find the statistical significance of our data, we have used Fisher's exact test, which is useful for observations classified in two ways. The test analyses contingency tables, which is a matrix displaying frequency distributions of variables as shown in Table 3.1 with an example of the number of people from Norway and England watching or not watching Doctor Who.

| | Watches Doctor Who | Do not watch Doctor Who | Total |
|---|---|---|---|
| England | 20 | 30 | 50 |
| Norway | 35 | 20 | 55 |
| Totals | 55 | 50 | 105 |

Table 3.1: Example of a contingency table of viewers of Doctor Who from Norway and England.

Fisher's exact test takes a matrix like this as an input, and calculates as shown in equation 3.1.

$$p = \frac{\binom{a+b}{a}\binom{c+d}{c}}{\binom{n}{a+c}} \tag{3.1}$$

Where $a, b, c$ and $d$ is the observations in our matrix. Taking Table 3.1 as an example, the input to the equation will be that $a = 20$, people from England watching Doctor Who, $b = 30$, Englishmen not watching Doctor Who, $c = 35$, Norwegians watching and $d = 50$, Norwegians not watching Doctor Who. $n$ is the total of individuals represented in the table, shown in the lower right corner, in our case $n = 105$.

The output of the test is the probability $p$ of our null hypothesis $H_0$ being true. The smaller the p-value is, the more evidence we have against $H_0$. The result of our example will then be $p = 0.996$, being strong evidence *for* our $H_0$, meaning that a person from England is more likely to watch Doctor Who than a person from Norway.

## 3.5 Evaluation criteria

In this section we will look into different criteria for student assignments we are going to evaluate. Some criteria are specific to the assignment given, some general to all assignments. For each set of evaluation criteria, there are both a reason for why it being a criteria and if it is relevant for the analysis in this thesis. We have divided the criteria into categories and not after the order they appear in the assignment description.

### 3.5.1 General criteria

The first category of criteria is the general ones. These are based on widely used code principles, such as clean code, narrowed down to the range of the university course. Table 3.2 shows criteria not specific to this assignment, but general for all programs written by students at this level. The general criteria for deliveries in the course states that all delivered programs must compile and run and we list this as a possible criteria to check for (table 3.2, criteria 1). Since this assignment is a couple of months into the course, we assume that the students have learned this by now and also make the argument that this criteria does not give us much relevant input about pointer comprehension and therefore exclude the compilation criteria from our analysis. It is important to write reusable, *general-purpose*, programs so that we can use it the next time we build a similar program (Drake, 2006). We find this criteria hard to check since the assignment description is very specific about what to implement, so we leave criteria 6 and 7 out of our analysis.

We want to check whether the students are able to abstract the program into reasonable classes, methods and code blocks, criteria 3. By code blocks we mean the content of if-tests, loops, try-s and such. In order to write good code it should be divided according to the functionality, the functions should be short and do one thing.

> "The first rule of functions is that they should be very small. The second rule of functions is that *they should be smaller than that*." (Robert C. Martin, 2008)

This is an important step in order to make general-purpose functions and objects, so in a way this includes some of the idea behind criteria 7 from Table 3.2.

| # | What | Comments | Relevant |
|---|---|---|---|
| 1. | Program free for compilation errors | This is expected to be learned in previous courses, thus not relevant for us. | ✗ |
| 2. | Program can handle null pointers without crashing | It's considered relevant in this course. We believe we can map some miscomprehensions of basic concepts. | ✓ |
| 3. | Division of tasks into own methods, short code blocs, reusable code | This can say something about the students level and comprehension of the whole program and its functionality. | ✓ |
| 4. | Reasonable use of variables | Includes both declaring variables in smallest score sensible and not declaring unnecessary many. Can indicate the students understanding of functionality. | ✓ |
| 5. | Reasonable variable, class and method naming | Bad or meaningless names of variables and such can give away if the student does not understand its function. | ✓ |
| 6. | The generic data structures are re-usable for other programs | Difficult to test in this program since the assignment description is too specific to give much freedom to make choices. | ✗ |
| 7. | The data structure interfaces are re-usable for other programs | Difficult to test in this program since the assignment description is too specific to give much freedom to make choices. | ✗ |
| 8. | User interface tolerates wrong user input | Errors at user input can be a result of the student not thinking about user friendly programs, but also lack of understanding. User-friendliness is not relevant for us. | ✗ |
| 9. | Program uses Java's built in classes and methods when possible | The assignment description makes this point irrelevant. | ✗ |

Table 3.2: General evaluation criteria

### 3.5.2 Criteria specific to the assignment

In table 3.3, 3.4, 3.5, 3.6 and 3.7 we have listed the criteria specific to the assignment. The criteria is numbered to easily refer to them and has a comment for why its a criteria. Some are retrieved directly from the assignment description, while others are our interpretation of it. From these criteria, we have chosen some to look into in more detail. For further information about what is directly from the text and what is our interpretation, see Appendix B.

The assignment specific criteria is categorised by whether they describe the class hierarchy, data structure or the program functionality. The criteria listed do all describe the program, but most of them can be applied to both program and diagram, the details about this will be listed more detailed at the end of this chapter. We have not listed the most basic criteria from the assignment description that do not play an essential role in the program, such as persons having a name.

Some of the criteria is applied to both code and diagrams, this will be further explained later in this chapter.

| # | What | Comments | Relevant |
|---|---|---|---|
| 10. | Class Doctor has a container for prescriptions | Important to the data structure in this assignment | ✓ |
| 11. | Implemented class for medicine | Including this in order to check for it being abstract and to check for its subclasses. | ✓ |
| 12. | The medicine class is abstract | Should test whether the student understands the concept of abstract classes | ✓ |
| 13. | Implemented interfaces for pills, ointment and injection. Implemented by the subclasses of medicine. | Implementation of interfaces is important in a program this big. | ✓ |

Table 3.3: Evaluation criteria for the class hierarchy and structure.

| # | What | Comments | Relevant |
|---|---|---|---|
| 14. | Subclasses of medicine for 3 types of drugs, that extends the interfaces for what form the drug comes in. Total of 3 subclasses with each 3 subclasses (11 in total) or 9 subclasses. | Creating subclasses is important to understand and make use of in a program of this size. | ✓ |
| 15. | Subclasses of Prescription for blue and white prescriptions | Not necessary to test subclasses again. | ✗ |
| 16. | Class Doctor implements Comparable with itself and Equal | We want to check how the students are comparing objects. | ✓ |
| 17. | Interfaces describes doctors with agreement with the state. | Do not need more test for implementation | ✗ |
| 18. | Doctors with specialisation is implemented as a subclass of Doctor | Do not need to test more subclasses | ✗ |
| 19. | Have implemented class Person | A very basic class | ✗ |
| 20. | Abstract class Person with subclasses for man and woman. | Not necessary to test subclasses or abstract classes again. | ✗ |

Table 3.4: More evaluation criteria for the class hierarchy and structure.

| # | What | Comments | Relevant |
|---|---|---|---|
| 21. | Generic interface AbstractTable | Generic types is a difficult subject, but we chose to not go into this in our thesis because it does not say much about comprehension of pointers, which we want to study. | ✗ |
| 22. | Generic interface AbstractSortedSimpleList containing elements of Comparable and Equal | Same reason as number 21. | ✗ |
| 23. | Generic class Table, implementing AbstractTable | Same reason as number 21. | ✗ |
| 24. | The class SortedSimpleList is generic and implements AbstractSortedSimpleList | Same reason as number 21 | ✗ |
| 25. | Implemented class SimplePrescriptionList | Central data structure class in this assignment | ✓ |
| 26. | Class SimplePrescriptionList is abstract | We include this criteria to find out something about the students understanding of what's the idea of its subclasses. | ✓ |
| 27. | OldestFirstPrescriptionList is a subclass of SimplePrescriptionList | Important part of the data structure we want to study in this assignment | ✓ |
| 28. | YoungestFirstPrescriptionList is a subclass of SimplePrescriptionList | Important part of the data structure we want to study in this assignment | ✓ |
| 29. | Both lists has Node classes | We say both instad of all because it is not necessary to write Node classes in the subclasses of SimpleList. | ✓ |
| 30. | Implemented menu / user interface | Important if we want to see whether the students manage to use the data structures they have made | ✓ |

Table 3.5: Evaluation criteria for the data structure.

| # | What | Comments | Relevant |
|---|---|---|---|
| 30. | Class AbstractTable returns an iterator over itself | Implementation of iterators is not part of the course, so it is difficult for us to test students understanding of this. | ✗ |
| 31. | SimplePrescriptionList contains an iterator | Same reason as number 30 | ✗ |
| 32. | AbstractSortedSimpleList returns an iterator over itself | Same reason as number 30 | ✗ |
| 33. | The find-method in SimplePrescriptionList checks for null-pointers in order to not crash | Checks if the student manages to use the list and understand when it ends. | ✓ |
| 34. | The find-method SimplePrescriptionList throws an exception if the prescription is not in the list | Exception handling is not relevant for our study of data structures and pointers | ✗ |
| 35. | The iterator in YoungestFirstPrescriptionList starts with the newest inserted node first | We want to compare the code here with the drawn structure in the object diagram | ✓ |
| 36. | The iterator in OldestFirstPrescriptionList starts with the oldest inserted node first | We want to compare the code here with the drawn structure in the object diagram | ✓ |
| 37. | If the student don't use list tail in SortedSimpleList: The student checks for null in the methods for finding and inserting nodes. | Important to see how they handles the end of the list | ✓ |
| 38. | If SortedSimpleList uses list tail: The student checks if node equals list tail to find where the list ends | As number 37 | ✓ |
| 39. | Class SimplePrescriptionList has abstract insert-method or no insertion method | Included to check how the student will implement this super class and its subclasses. | ✓ |

Table 3.6: Evaluation criteria for program functionality.

| # | What | Comments | Relevant |
|---|---|---|---|
| 40. | The list objects has reference to next element in the list | In order to create a list | ✓ |
| 41. | The user interface has functionality for creating and inserting new medicine, doctors, persons and prescription | Interesting to see if the student manages to use the data structure | ✓ |
| 42. | The user interface has functionality for reading and writing all data in the program to file | Not in our scope in this thesis | ✗ |
| 43. | The user interface can search the data structure for persons and doctors | Relevant to see if the student manages to use the data structure, but we don't need to check this point and number 41 | ✗ |
| 44. | The user interface uses the iterators in the containers | How to use iterators is relevant on curriculum. Interesting to see in the insertion methods described in criteria number 41. | ✓ |
| 45. | The method for finding an element in SimpleSortedList uses the method "same" in the interface Equal to compare | This is interesting, but we decided to focus on number 46 instead since both is not necessary. | ✗ |
| 46. | The method for inserting an element in SimpleSortedList uses the method "compareTo" in the interface Comparable to compare | Checking the comprehension of the difference between object equality, pointer equality and whether the students understands what element they are comparing. | ✓ |

Table 3.7: More evaluation criteria for program functionality.

### Abstract classes

We have chosen to include abstract classes in our study because this is a concept that is not necessary to include in order for the program functionality to be as expected, but therefore it could give us a indication of those students that have an understanding of these two classes role in the program.

#### Discussion of abstract insertion method in SimplePrescriptionList

Assignment text does state that the class SimplePrescriptionList should have functionality for insertion and searching, still we have chosen to see whether the insertion method is abstract. The thought behind this is that we hope to find out whether the students are writing duplicate code. To write code for the same functionality more than once is considered code smell, meaning it is bad code (Beck and F. Martin, 1999). Also this criteria combined with whether SimplePrescriptionList is abstract can be compared to how they draw the prescription and list objects in the object diagram. If this is abstract, the SimplePrescriptionList should not be present in the diagram.

#### Discussion of list tail

Some students create a list head node and set the list tail-pointer to point at the head node. This could be an OK solution because we can easily check if this is the first time we've seen this node in this iteration or if it is the second time. We have chosen to look at the cases where list tale does not point at list head, to see how they terminate their lists, because we have found no example of students having list tail pointing at list tale and checking whether the current node is a list tale that they have seen the second time. See figure 3.5 for an example. In these cases there would be more interesting to see if they checks for null pointers in the list, to terminate the loop.

#### Using iterators without checking for null

The lecture notes is writing for-each-loops and using iterators without checking if the current element is null. This is included in our analysis because it the students are not expected to have implemented the functionality of their iterators themselves, nor to understand it thoroughly, meaning if they check for null when iterating, it could indicate understanding of that null pointers can be present even when not expected.

### 3.5.3 The relevant criteria for the code

We will refer to the class SimplePrescriptionList as SimpleList, YoungestFirstPrescriptionList as YoungestFirstList or YoungestFirst, OldestFirstPrescriptionList as OldestFirstList or OldestFirst and SortedSimpleList as SortedList.

Table 3.8 and 3.9 describes the criteria that we have found the most relevant to study in detail, marked with ✓in the tables 3.2, 3.5 3.6 and 3.7. We picked criteria we think cover a variety of topics and problem areas connected to pointer and data structure comprehension. The criteria were given a maximum score of either 1 or 2 in order to whether they are "small" tasks or consists of a larger task, like "Both lists has Node classes".

```
\\ with list head and list tail:
Node listHead = new Node("listHead");
Node listTail = new Node("listTail");

// some function that is traversing through the list
void someFunction() {
  if (listHead.next == null) return;

  Node current = listHead.next;
  while (current != listTail) {
    ...
    current = current.next;
  }
}
...

\\ second alternative:
Node first; // pointer to first node in list or null if empty list

// some function that is traversing through the list
void someFunction() {
  Node current = first;
  while(current != null) {
    ...
    current = current.next;
  }
}
```

Figure 3.5: Example of the two “ideal” solutions of iterating through a list

The general criteria, number 2 - 5, in table 3.8 are weighted to 2 because they include code all over the assignment. Criteria like 25, 27 and 28 are taken from the assignment description and at this point in the semester the students should know how to implement classes and subclasses when they are asked to.

| # | **General criteria** | **Comments** | **Points** |
|---|---|---|---|
| 2 | Program can handle null pointers without crashing | | 2 |
| 3 | Division of tasks into own methods, short code blocs, reusable code | | 2 |
| 4 | Reasonable use of variables | | 2 |
| 5 | Reasonable variable naming | | 2 |
| **#** | **Data structure criteria** | **Comments** | **Points** |
| 11 | Implemented class Medicine | | 1 |
| 12 | Medicine is abstract | | 1 |
| 13 | The subclasses of Medicine implements the 3 form interfaces | | 2 |
| 14 | Medicine has subclasses for type and form | 3 subclasses of Medicine that each has 3 subclasses, as shown in figure 3.1 *or* 9 subclasses of Medicine as shown in 3.2 | 2 |
| 25 | Implemented class SimplePrescriptionList | | 1 |
| 26 | SimpleList is abstract | | 1 |
| 27 | OldestFirst is a subclass of SimpleList | | 1 |
| 28 | YoungestFirst is a subclass of SimpleList | | 1 |
| 29 | Both lists has Node classes | We say both instad of all because it is not necessary to write Node classes in the subclasses of SimpleList. | 2 |

Table 3.8: Criteria relevant to the code. Listing general and classes and data structure criteria.

Since creating iterators is not a topic in the course, we chose to reformulate the criteria 39 and 40 from table 3.6 to look at the insertion method instead. Note that the iterator solution is counted as an equally good solution in our study thus resulting in the same amount of points.

| # | Criteria to the functionality | Comments | Points |
|---|---|---|---|
| 34 | The find-method in SimpleList checks for null-pointers in order to not crash | | 2 |
| 36 | The insert-method in YoungestFirst inserts nodes at the front of the list | OR the iterator for the list starts with the newest inserted node (last in first out) | 2 |
| 37 | The insert-method in OldestFirst inserts nodes at the end of the list | OR the iterator for the list starts with the oldest inserted node (first in first out) | 2 |
| 38 / 39 | SortedList has list tail and uses it to find end of list *or* has not list tail and checks for null | | 2 |
| 40 | SimpleList has abstract or none insertion method | | 2 |
| 41 | Both list objects has references to next element in the list | | 2 |
| 42 | The user interface has functionality for creating and inserting new medicine, doctors, persons and prescription | | 2 |
| 45 | The user interface uses the iterators in the container | | 2 |
| 47 | The method for inserting an element in SimpleList uses the method “compareTo” in the interface Comparable to compare elements | | 1 |

Table 3.9: Criteria relevant to the code. Criteria for the program functionality.

### 3.5.4 The relevant criteria for the diagrams

The assignment asks the students to draw a class hierarchy of the class structure we have described in table 3.3 and a object diagram of how the lists and tables should work and illustrate with some inserted objects in all containers. Table 3.10 contains the criteria for evaluating the diagrams, marked with ✓in table 3.3. We chose these mainly because they can be compared to the delivered code and that way we hope to find out something about the students comprehension of their program

components and functionality of the data structures. The criteria for evaluating the object diagrams is also from the functionality criteria, but formulated to describe a diagram.

| # | The class hierarchy | Comments | Points |
|---|---|---|---|
| 1.1 | The students have drawn a class hierarchy | Does not have to be good or complete | 1 |
| 2 | Contains class for medicine | | 1 |
| 6 | The medicine class is abstract | | 1 |
| 4 | Doctor implements Comparable and Equal | | 2 |
| 5 | Class Doctor has a container for Prescriptions | | 1 |
| 7 | Interfaces for pills, ointment and injection. Implemented by the subclasses of medicine. | | 2 |
| 8 | Subclasses of medicine for 3 types of drugs, that extends the interfaces for what form the drug comes in. | Total of 3 subclasses with each 3 subclasses (11 in total) or 9 subclasses. | 2 |
| **#** | **The object diagram** | **Comments** | **Points** |
| 1 | The student has drawn an object diagram | Checking whether the delivered diagram is to poor to analyse. Does not have to be good or complete. | 1 |
| 2 | The object diagram is readable / understandable | | 2 |
| 3 | The object diagram is an object diagram | Checking whether it is according to the specifications of an object diagram. | 2 |
| 4 | The object diagram is according to the assignment description | We need to check if the diagram illustrates the criteria described in the assignment description. | 2 |
| 5 | Shows that SortedList has a pointer to a list of Doctor objects | | 2 |
| 6 | Shows that YoungestFirst has pointers to Prescription objects | | 2 |
| 7 | Shows that OldestFirst has pointers to Prescription objects | | 2 |

Table 3.10: Criteria relevant to the diagrams

# Chapter 4

# Results

In this chapter we will present the results found in our analysis. The highest possible score for the program code were 37 points and for diagrams 23 points, making the highest achievable total score 60 points. In addition, it was possible to score 9 points at the correlation between code and diagram, but this number is not included in either of the total, code or diagram score. Instead, this is used as an external measure after dividing the students in three categories according to their scores. The correlation shows the average score on the correlation between code and diagram and does, only looking at to what degree they show the same program or not, not taking into account the scores on anything else and is not dependent on whether the solution is correct. The diagram scores checks for more criteria than can be compared directly to the code (like whether the diagram is readable), thus it is possible to score low on the diagrams but high on the correlation. We analysed the average scores for all students and for the best and the poorest thirds of the data set.

## 4.1 Scores from the entire set of students

We analysed in total 57 deliveries. Table 4.1 contains an overview of how many deliveries that contained what diagrams. As we can see, 14% did not deliver anything and 68% did contain both diagrams. This will have an impact on the results in this chapter that is looking at the scores of all deliveries.

None of the deliveries in our data set got maximum score, neither on total, code or diagram score.

1 of the deliveries was left out of the comparisons, since this delivery was almost empty, only containing class definitions with some declared fields, but not any methods. Since this is so poor it is considered an outlier in the data set, we have chosen to exclude it.

### 4.1.1 Distribution of the student groups

The figures 4.1, 4.2 and 4.3 shows the distribution of the students points in percentage of maximum possible score. Respectively they show

| | | |
|---|---|---|
| Class hierarchy | 47 | 80% |
| Object diagram | 40 | 68% |
| Both diagrams | 39 | 68% |
| Only class hierarchy | 8 | 14% |
| Only object diagram | 1 | 1% |
| No diagrams | 8 | 14% |

Table 4.1: Overview of how many student deliveries contained what diagrams, not including the almost empty delivery. The two first rows contains some of the same students, the last four rows show the details. One might notice that the sum of the percentages of the last four rows is not 100, this is a result of the percentages being rounded up or down to nearest integer. The sum of the numbers in the second column for these four rows still is 56, the total number of student groups included in our analysis.

the distribution of the student groups according to total score, code score and the latter show the distribution of the diagram score, only taking into account those that delivered both and not the 17 that did not. From figure 4.1 we can see that no one had scored less than 30% of total maximum score and that more than half of the deliveries scored above 50%. In figure 4.2 we see a quite normal distributed graph, showing that most students scored between 60% and 80%. Figure 4.3 shows a quite equal distribution between 20% and 90 %. We used these distribution graphs to divide the deliveries into the three categories for later comparison with the category with the highest scores and the lowest scores.

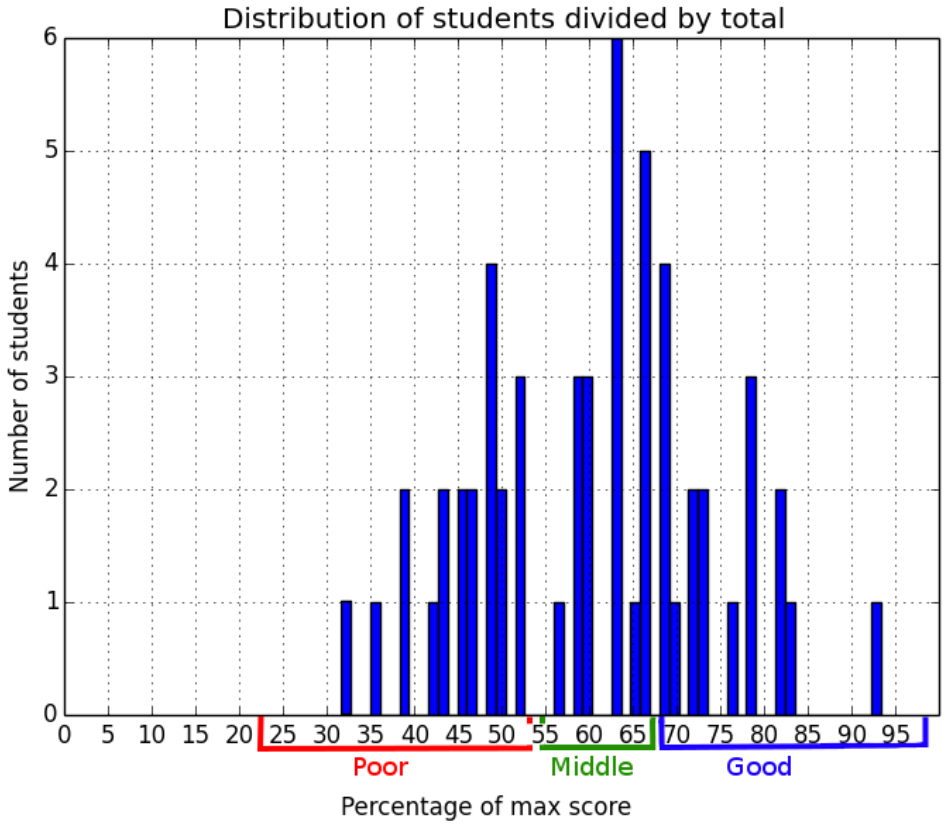


Figure 4.1: This figure shows the distribution of the total scores. The x-axis is in percentage of maximum total score and the y axis shows number of students.

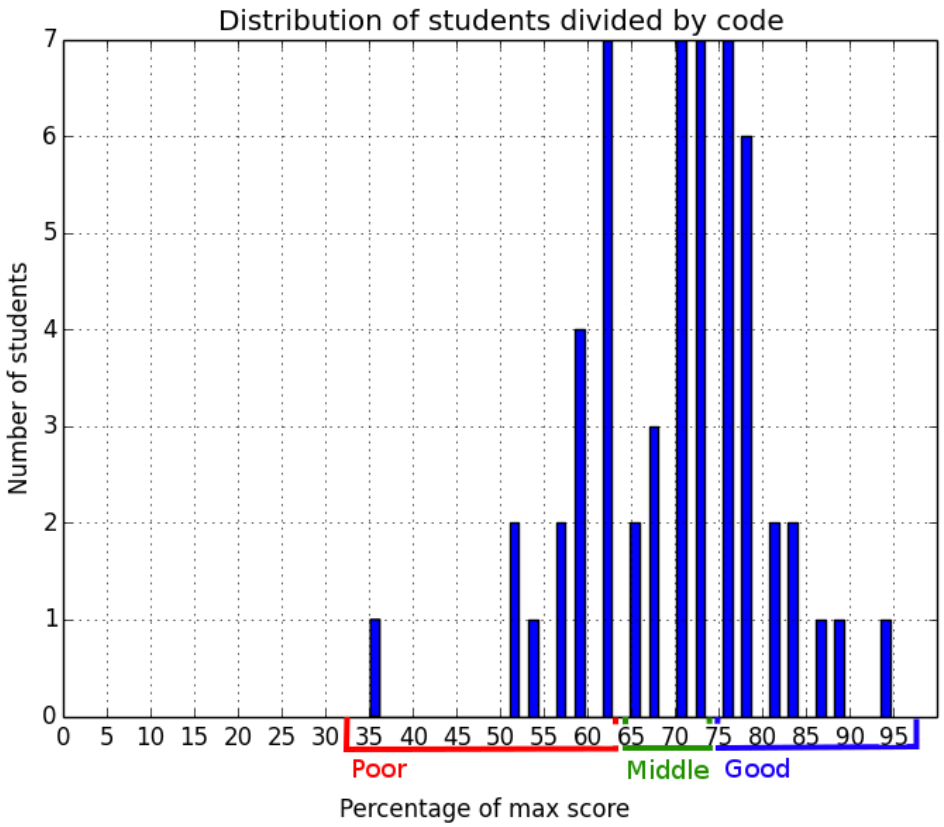


Figure 4.2: This figure shows the distribution of the code scores. The x-axis is in percentage of maximum code score and the y axis shows number of students.

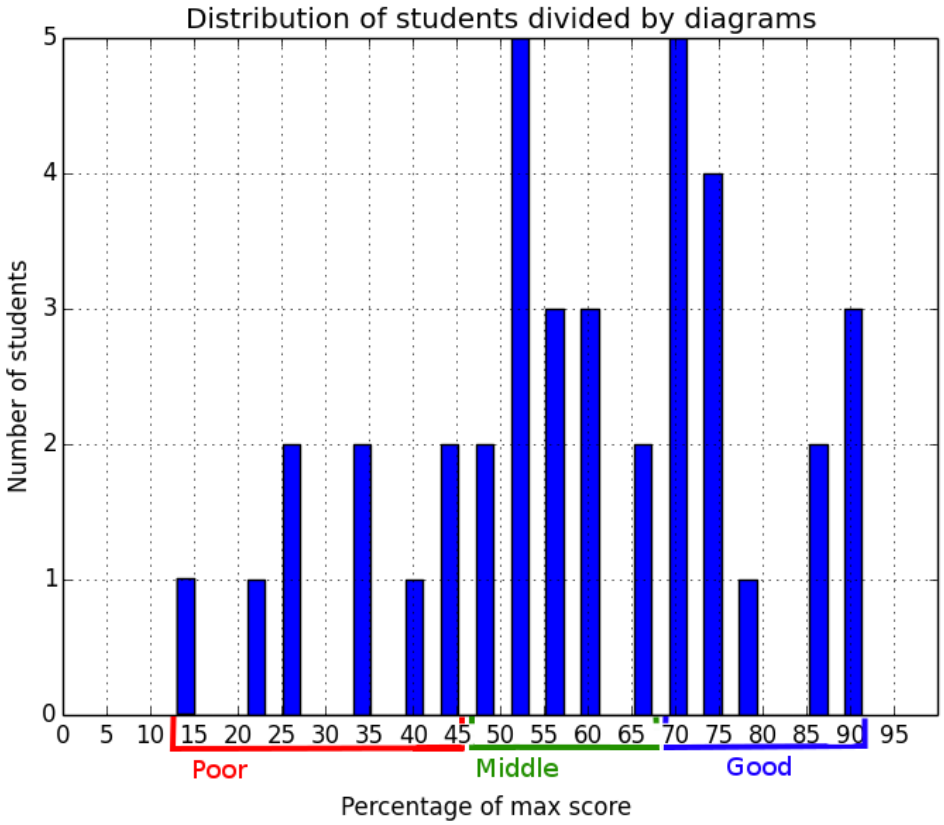


Figure 4.3: This figure shows the distribution of the diagram scores. The x-axis is in percentage of maximum diagram score and the y axis shows number of students. Only the student groups containing both diagrams are represented.

### 4.1.2 Average scores for all students

Before comparing the categories as divided in the previous section, we look at the average scores for the entire data set. The "diagram vs code" point in Figure 4.4 shows the correlation between diagrams and code. High correlation does not necessary indicate good code or good diagrams, just that they illustrate the same. It can also state that a feature, e.g. a class, is *not* present in neither of code diagram, thus those that did not implement every part of the program nor draw them in the diagram will have a high correlation.

It is worth noting in figure 4.4 that the average score on the diagrams is 45% of maximum diagram score (total score on class hierarchy and object diagram), while the average on the correlation is almost 60%, meaning that the diagrams are quite poor but still reflect the main criteria from the code. If we only look at the deliveries containing at least one diagram, the average score on diagrams is 55%. The scores on the data structure and general criteria, Figure 4.5, is above 70% while the functionality score is less than 65%, suggesting that the functionality contains some of the more difficult

criteria. When we only include the students groups containing the class diagram, the average score on this area is 58%, about 10% higher than the avarege of *all* students shown in figure 4.5. Doing the same for the object diagrams, the score jumps from 42% up to 59%, meaning that the delivered diagrams is about equal in quality.

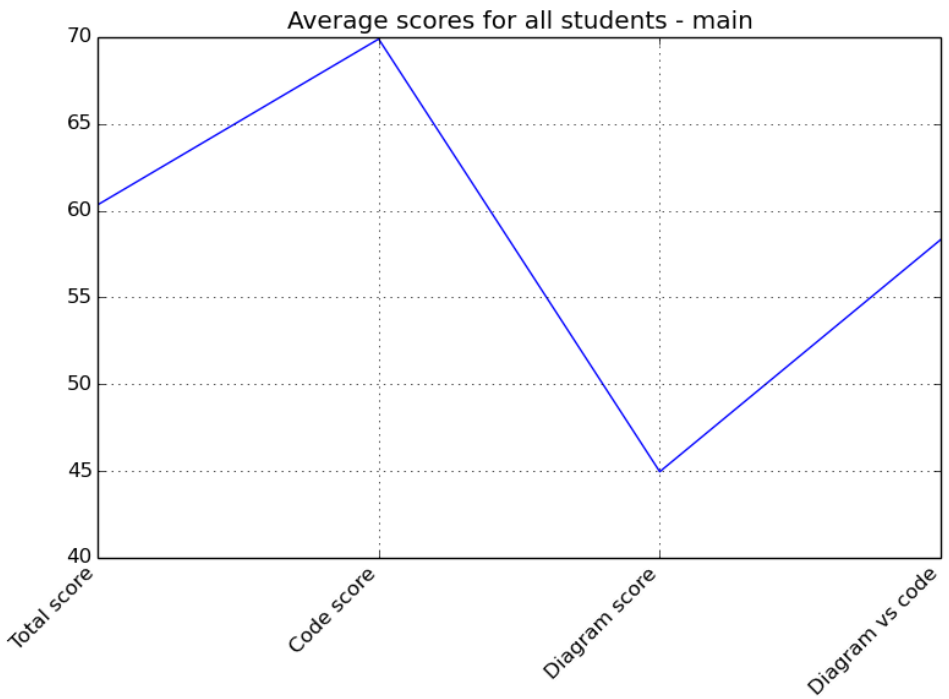


Figure 4.4: Average scores of all the deliveries in the data set. Showing scores in percentage of maximum score of total and code, diagram and the correlation between diagram and code.

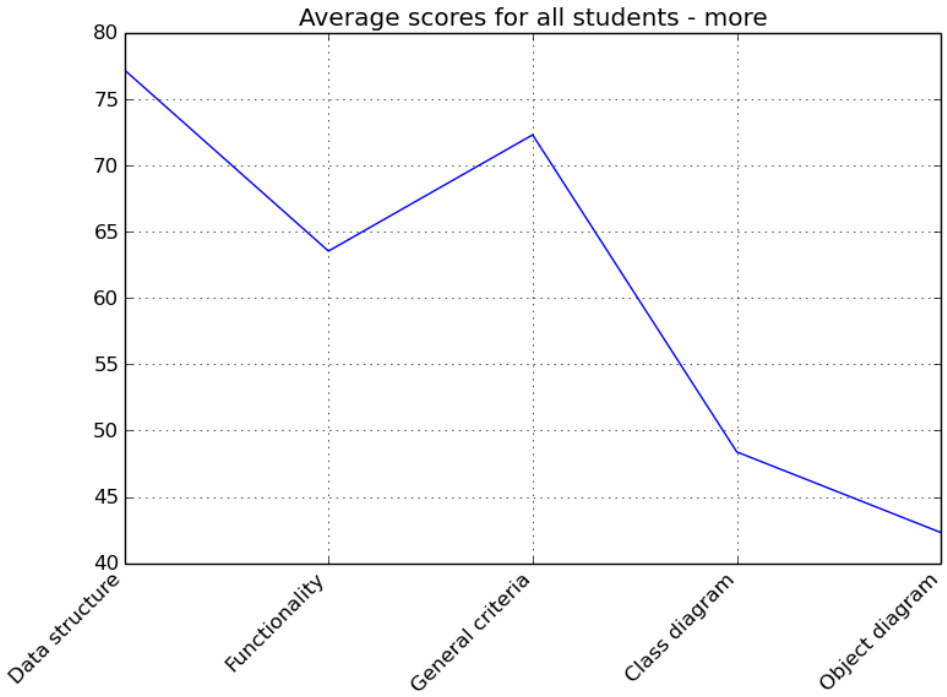


Figure 4.5: Average scores of all the deliveries in the data set. Showing the scores of all the main categories in percentage of the maximum scores.

In figure 4.4 and 4.5 we find average scores on the main areas. while figure 4.6, 4.7, 4.8, 4.9 and 4.10 shows the average scores in more detail.The criteria “Medicine has subclasses for type and form” is a summary for the two alternatives for class structure shown in figure 3.1 and 3.2 and is to be found both in the data structure and the class diagram criteria. In figure 4.6 and figure 4.9 we see that there is between 20 and 30% that managed to implement one of these alternatives in code and below 20% that managed to include it in the diagrams. The criteria “Object diagram is an object diagram” is from now called “Drawn an object diagram”.

We can see that the average score on “Reasonable naming” in figure 4.8 is quite high (more than 90%).

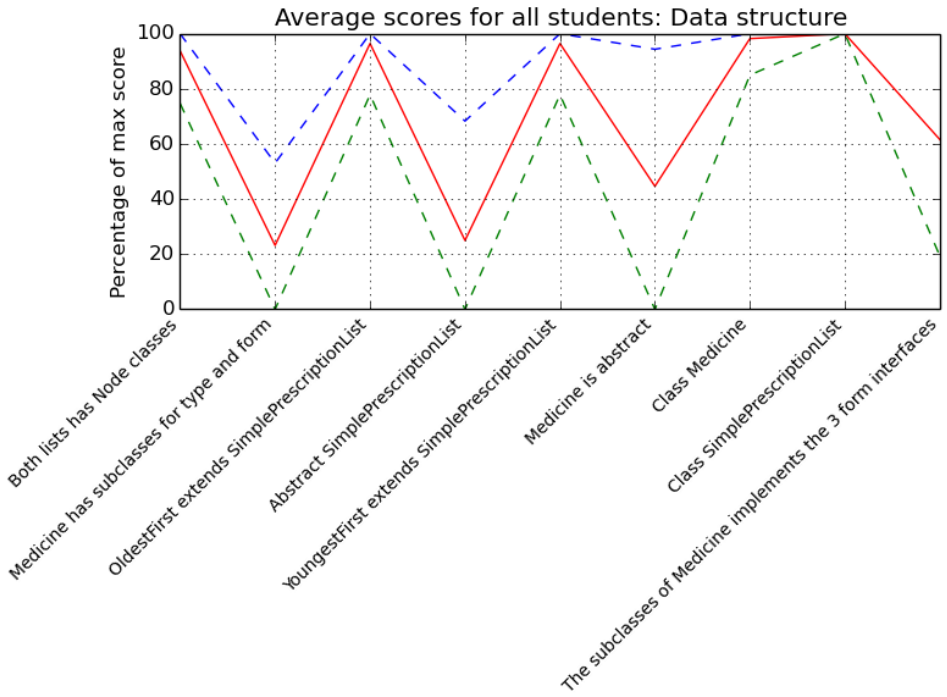


Figure 4.6: Details on the average scores on the data structure criteria. The solid line is the average score on the criteria on the x-axis while the dotted lines marks one standard deviation.

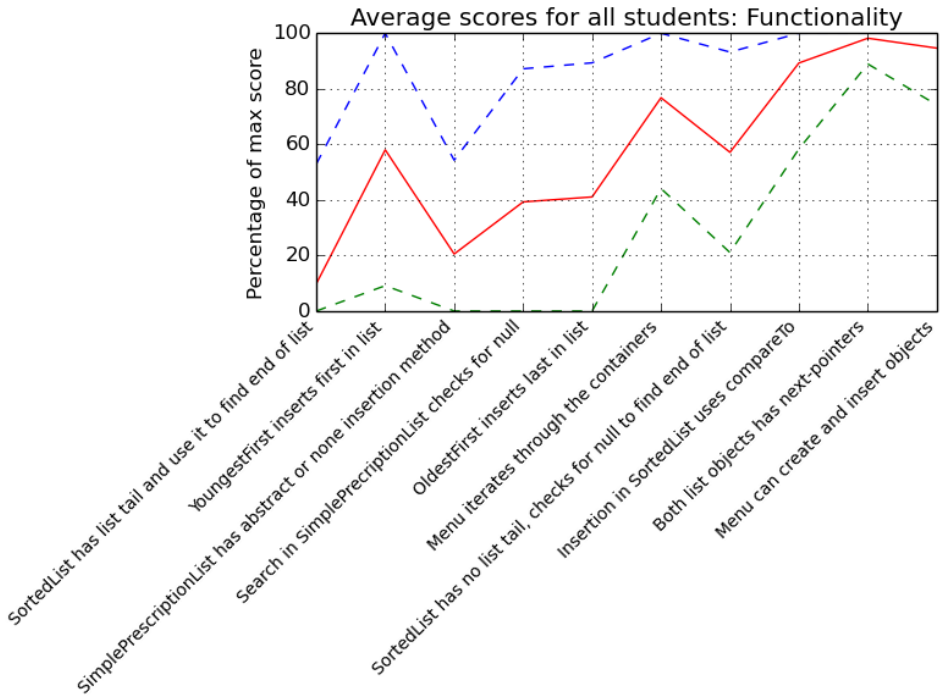


Figure 4.7: Details on the average scores on the functionality criteria. The solid line is the average score on the criteria on the x-axis while the dotted lines marks one standard deviation.

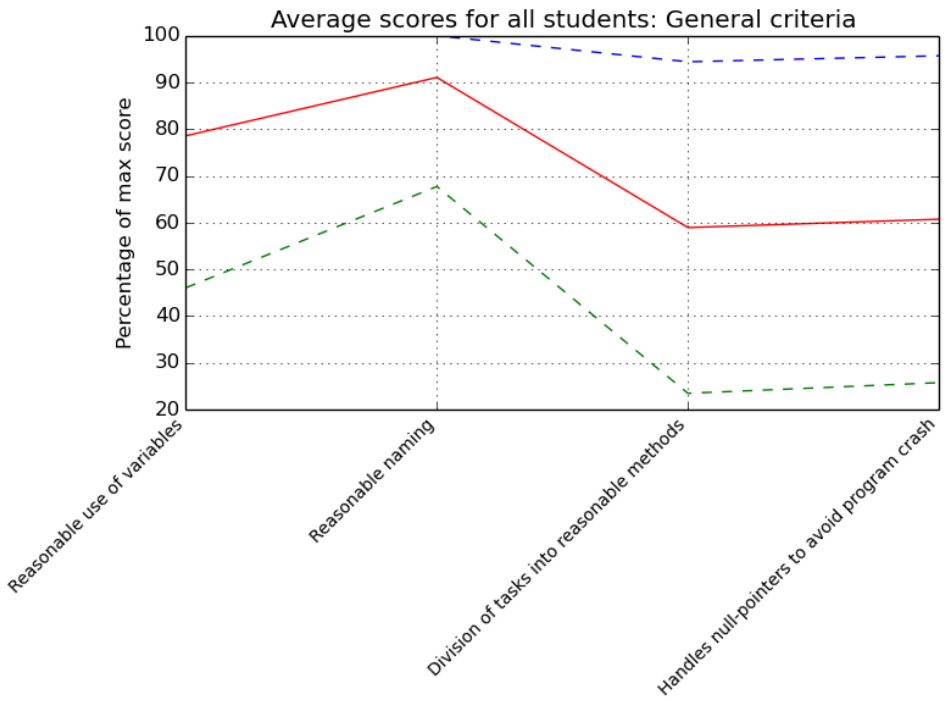


Figure 4.8: Details on the average scores on the general criteria. The solid line is the average score on the criteria on the x-axis while the dotted lines marks one standard deviation.

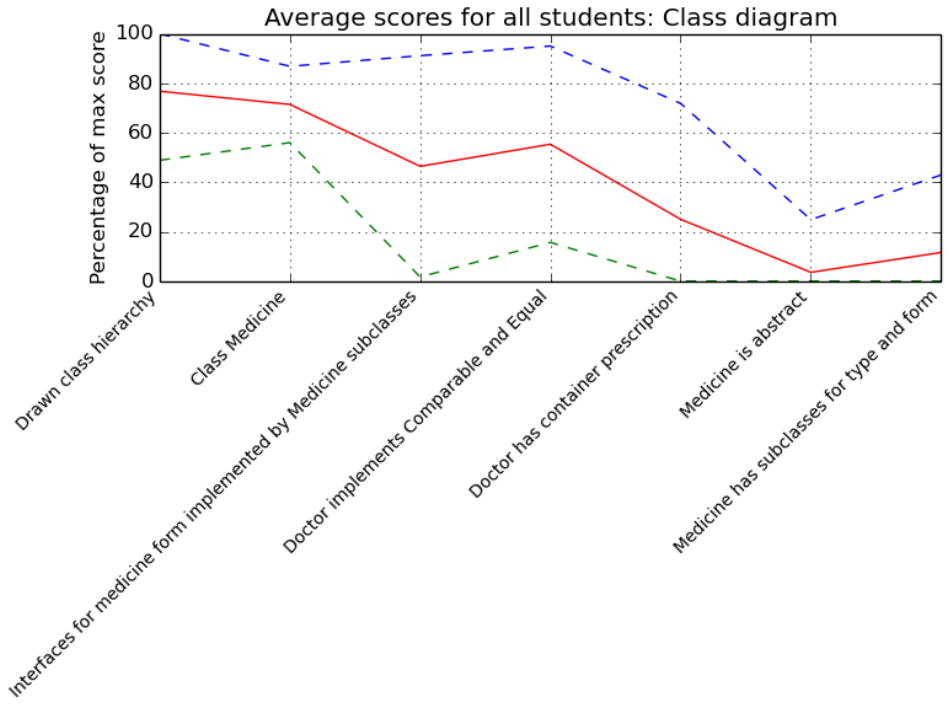


Figure 4.9: Details on the average scores on the criteria for the class diagram. The solid line is the average score on the criteria on the x-axis while the dotted lines marks one standard deviation.

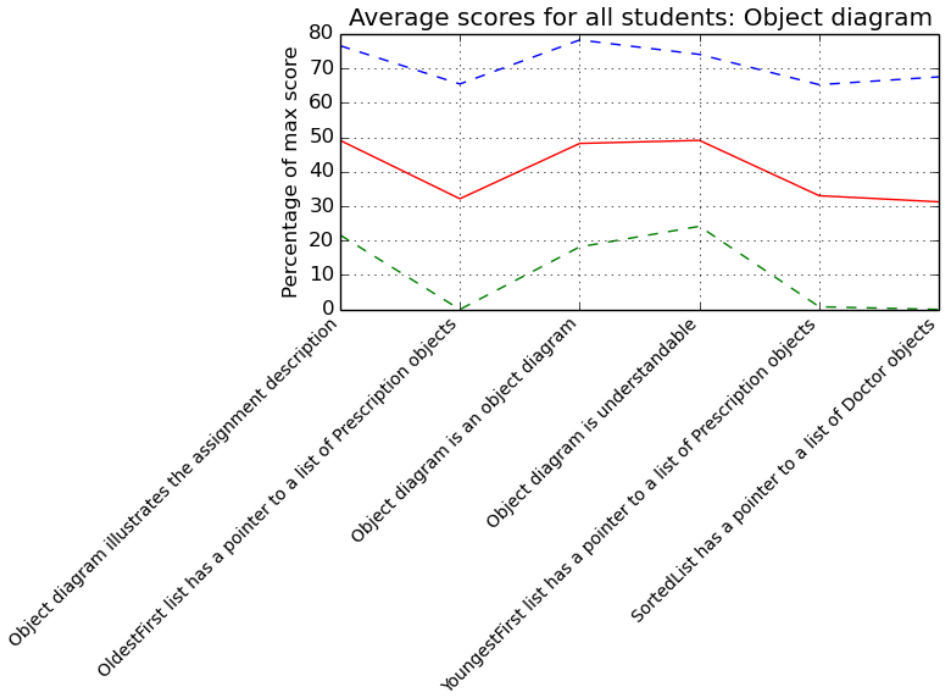


Figure 4.10: Details on the average scores on the criteria for the object diagram. The solid line is the average score on the criteria on the x-axis while the dotted lines marks one standard deviation.

## 4.2 Comparing the poorest and best thirds

As described in chapter 3.5, we analysed the students deliveries, gave them points for what they had achieved and categorised errors. We then were able to divide the deliveries into three categories (poor, average and best) according to their score on code alone, their diagram score and their total score on both. We have not focused on the middle third because we wanted to focus on finding out whether there was a significant difference between the best and the poorest students.

To calculate the significance of the results at the individual criteria, we applied this null hypothesis and alternative hypothesis to each of the criteria:

$$H_0 : \mu_b = \mu_p,$$
$$H_a : \mu_b > \mu_p,$$

Where $\mu_b$ is average of best category and $\mu_p$ is average of poorest category for that exact criteria. The p-values used to confirm or reject $H_0$ is displayed along the x-axis in the plots in this chapter. We will be using a significant level $\alpha = 0.05$.

The next sections will show the details for the criteria with some significant differences. Appendix C shows the plots not included in this

| What | Poor | Middle | Best |
|---|---|---|---|
| Average null-pointer errors | 19% | 15% | 12% |

Table 4.2: Average numbers of errors for the groups divided by total score. The numbers show how big percentage of maximum possible numbers of errors the students got.

| What | Poor | Middle | Best |
|---|---|---|---|
| Delivered hierarchy diagram | 50% | 94% | 100% |
| Delivered object diagram | 18% | 89% | 95% |

Table 4.3: Percentage that delivered diagrams for groups sorted by total score.

section.

### 4.2.1 Divided according to total score

The numbers in table 4.2 and 4.3 are based on dividing the deliveries into thirds according to the total score (sum of both code and diagrams). There are a notable lower percentage of the poorest group that has delivered the diagrams, especially the object diagram.

In figure 4.11 we can see what percentage of the maximum score the average student group in each of the three categories scored. The thirds seems to “rise and drop” at the same places, but we can see that all thirds scored more on the code than the diagrams and more on the consistency between code and than on the diagrams them selves. More detailed scores is shown in figure 4.12, where we can see that the differences between the thirds is quite small at all three code areas and quite big on the diagram areas.

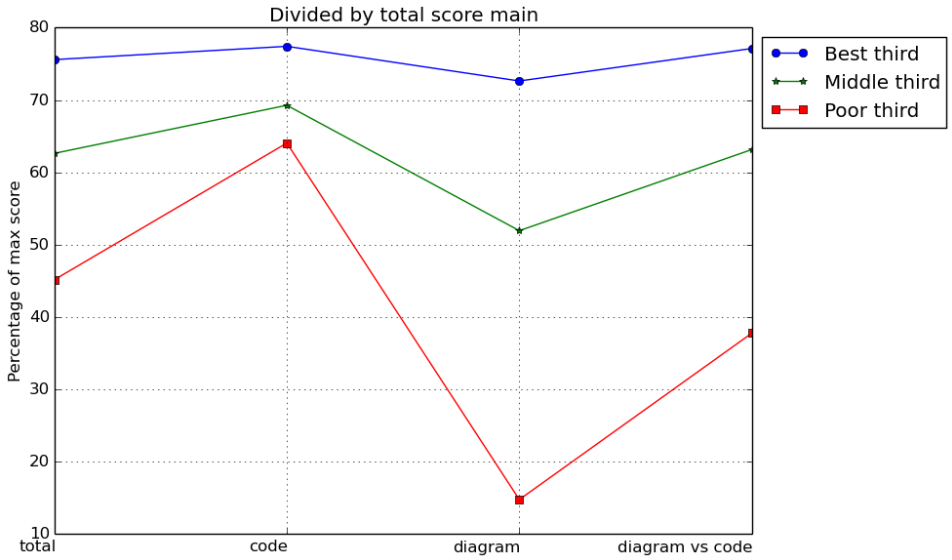


Figure 4.11: Average scores for total score

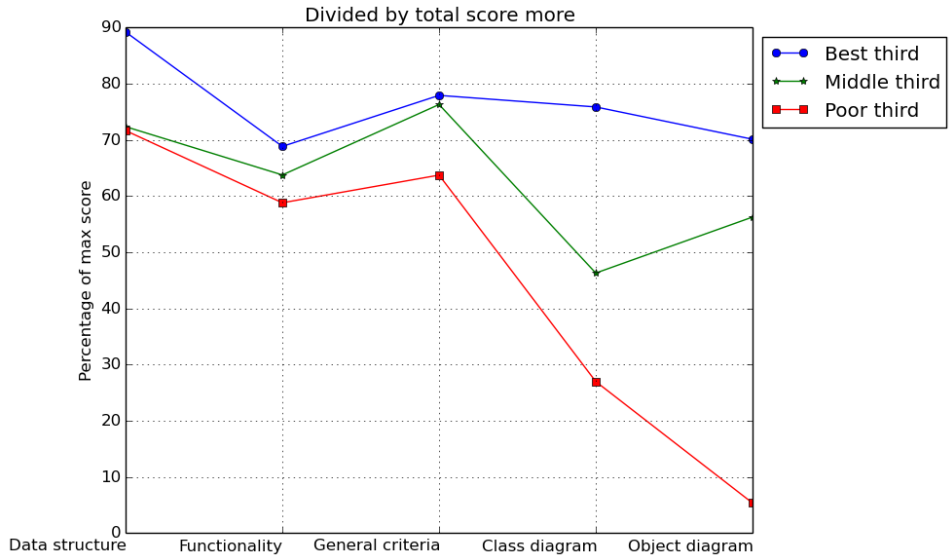


Figure 4.12: Average scores for total score

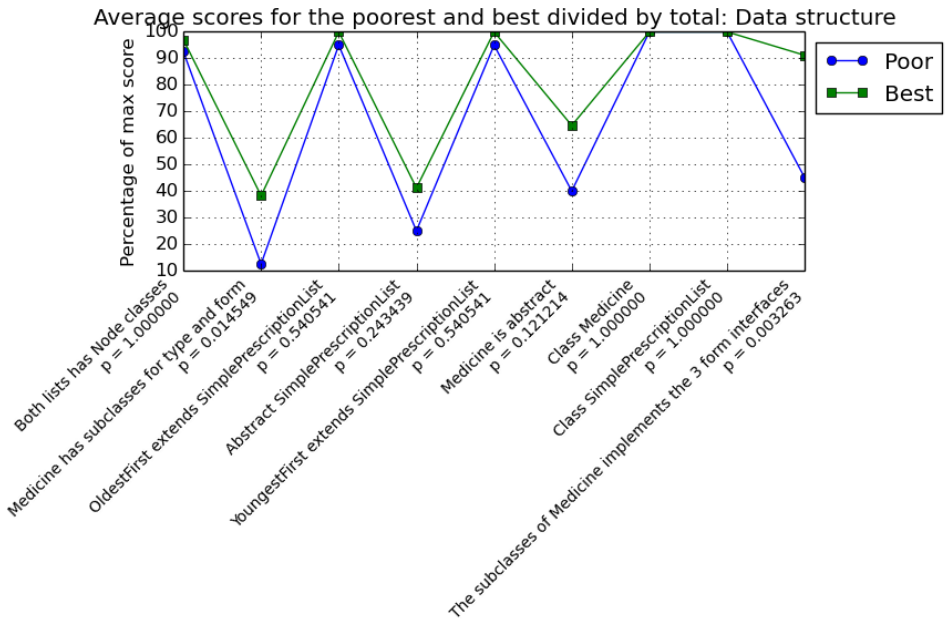


Figure 4.13: Average scores for all students data structure, students divided by total score.

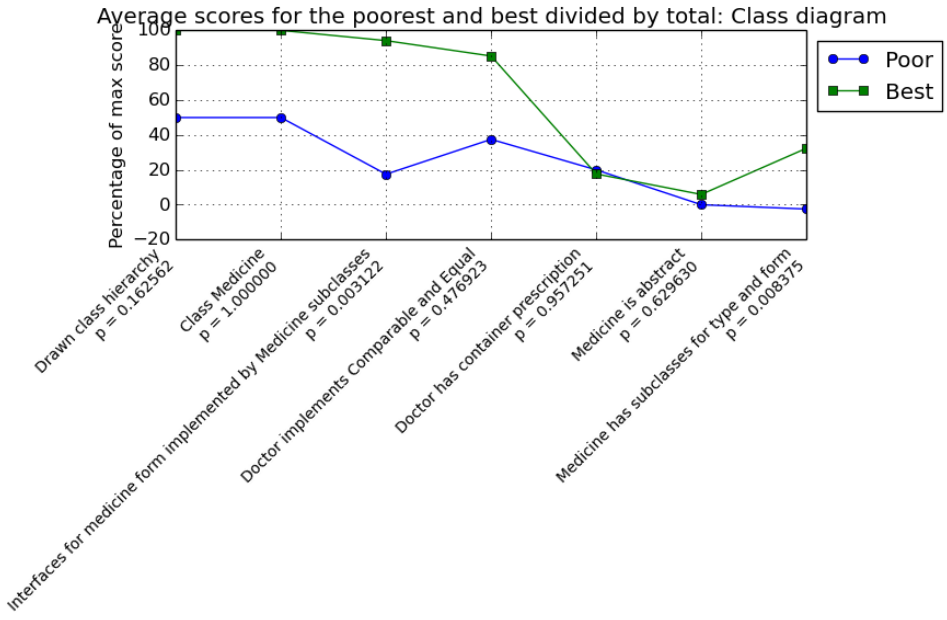


Figure 4.14: Average scores for all students class diagram, students divided by total score.

At data structure and class diagram, figure 4.13 and 4.14 we see by the p values that there is a significant difference for the criteria “Interfaces for medicine form implemented by Medicine subclasses” and “Medicine has subclasses for form and type”. For the object diagram, general and

| What | Poor | Middle | Best |
|---|---|---|---|
| Average null-pointer errors | 20% | 18% | 7% |

Table 4.4: Average numbers of errors for the students divided by code score.

| What | Poor | Middle | Best |
|---|---|---|---|
| Delivered hierarchy diagram | 88% | 84% | 80% |
| Delivered object diagram | 70% | 78% | 65% |

Table 4.5: Percentage that delivered diagrams for groups sorted by code score, equal group sizes

functionality criteria, however, we did not find any significant differences. This is suggesting that the total score does not say much about what makes a delivery good or poor.

## 4.3 Comparing poor and best code

In order to research whether there is a difference in the code quality and comprehension of the program at those students that succeed in creating diagrams, we calculate the same numbers for the deliveries divided by code score alone. Table 4.4 and 4.5 shows the average null-pointer errors for the three categories and the percentage that delivered the different diagrams. The poorest deliveries scored on average 57% of maximum code score and the best scored 80%. We still note the low score on diagrams for all three categories, between of 40% and 50%. From table 4.5 we can see that when we divide the thirds according to the code score, the distribution of the deliveries containing the diagrams were more equal than when divided by total score. In fact has a higher percentage of the poorer third delivered the diagrams than the best third.

Figure 4.15 and 4.16 displays the average scores for the poorest and best deliveries at the main areas. Interesting to see, in figure 4.15, is that the poorest student groups score almost 10% higher than the middle and best third at the correlation between diagram and code, even though they score lower on the diagrams them selves. As described in section 4.1, this score is not taken into account when dividing the student groups into categories, so it is possible that the poorest students have managed to recreate their program according to what they have illustrated in the diagrams while the best students have not. Recall that the lesser of the program implemented and drawn in the diagrams, the higher the correlation score will be.

The same figure shows an expected distribution of the code score, varying from an average of 80% for the best third and 57% score for the poorest.

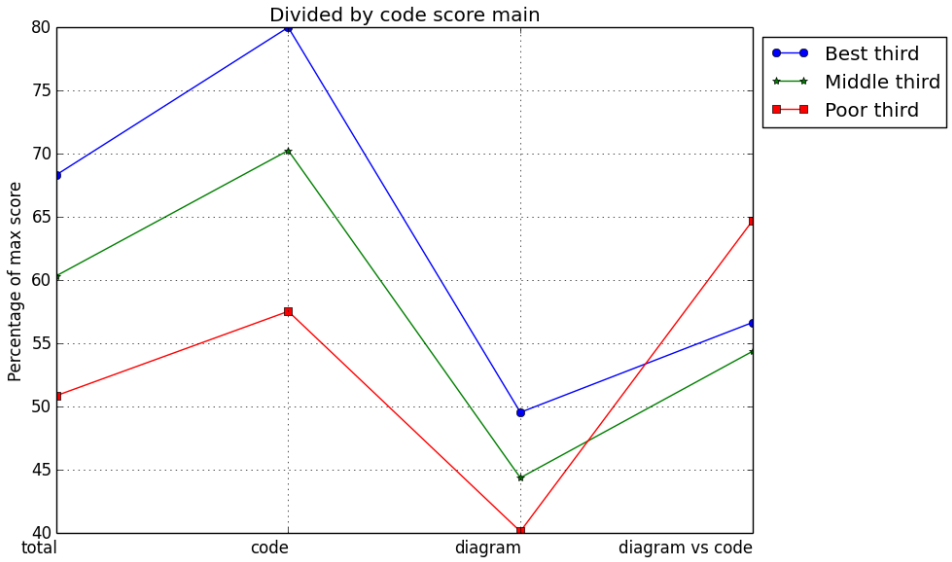


Figure 4.15: Average scores for code score

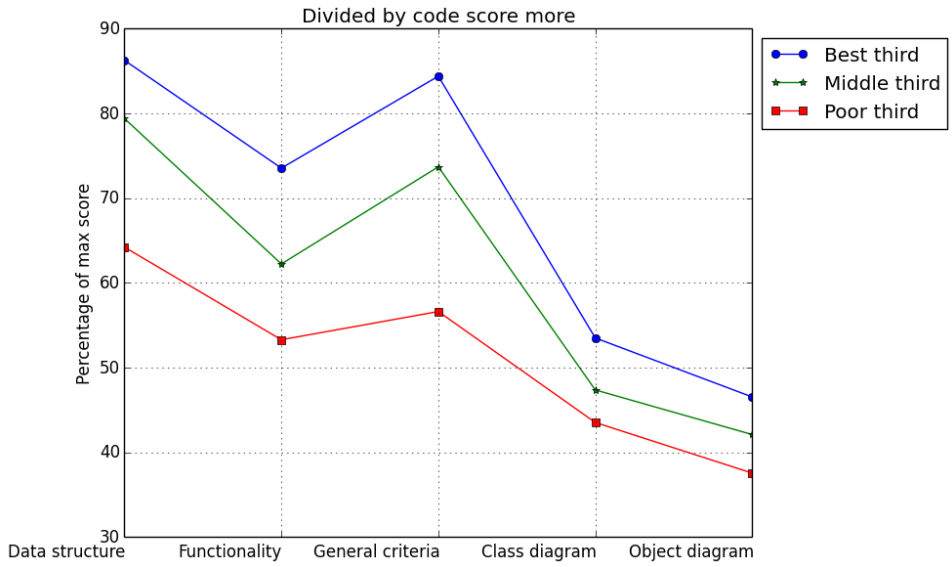


Figure 4.16: Average scores for code score

The detailed scores for the code is to be found in figure 4.17, 4.18, 4.19 and the details for the class hierarchy in figure 4.20.At the class and data structure criteria, figure 4.17 we find that the difference between the poorest and the best deliveries is not significant, except for when it comes to having the Medicine as an abstract class, having subclasses for medicine type and form and that they implement interfaces for medicine form. We also see that the average scores for the best and the poorest are a quite different for these two criteria. Except for some null pointer handling and the use of the compareTo-method for comparison, there are not much significant at the functionality in figure 4.18, but in general criteria in figure

4.19 we see from the p-value that the null-pointer handling for the entire program is not significantly different and that the average scores for the poorest and best are mostly “following” each other. There is, though, a quite significant difference between the best and the poorest at dividing the code into reasonable methods.

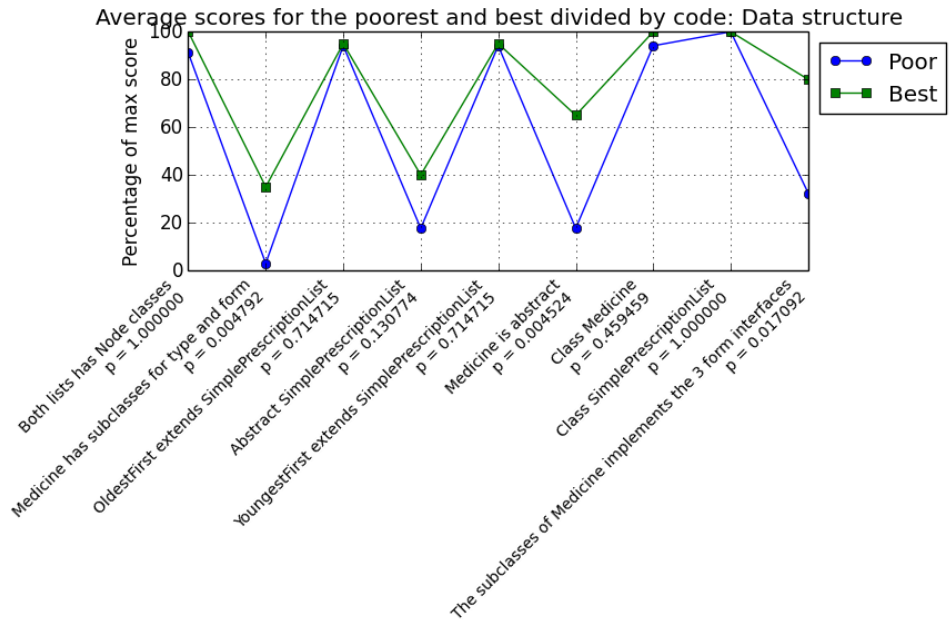


Figure 4.17: Average scores for the data structure, students grouped by code score.

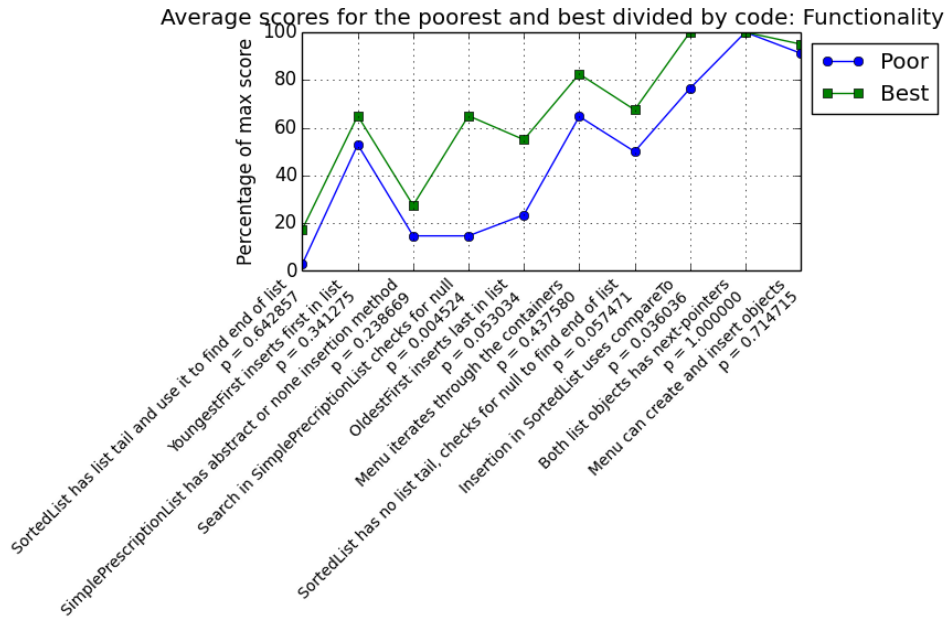


Figure 4.18: Average scores for the functionality, students grouped by code score.

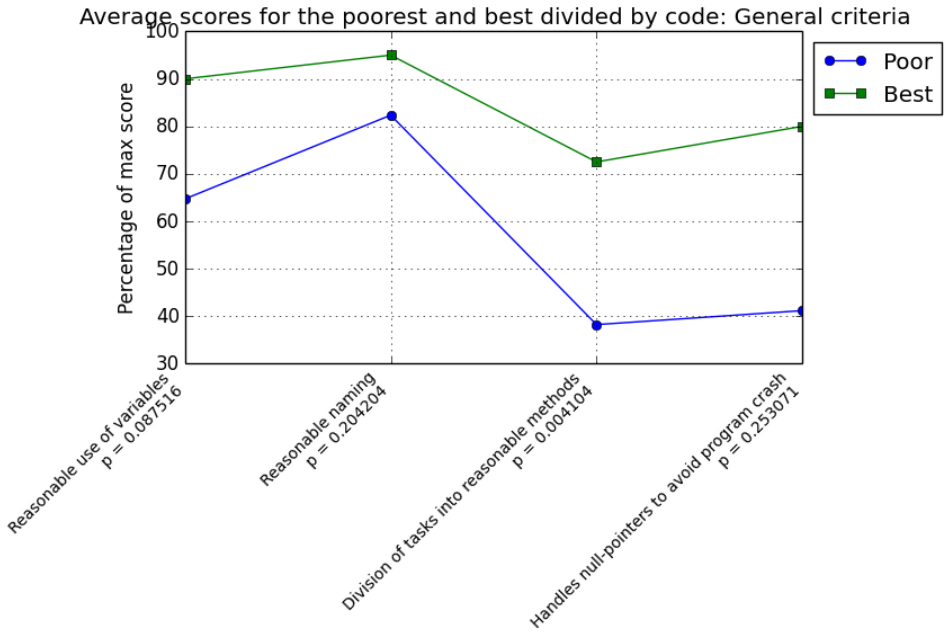


Figure 4.19: Average scores for the general criteria, students grouped by code score

| What | Poor | Middle | Best |
|---|---|---|---|
| Average null-pointer errors | 20% | 10% | 15% |

Table 4.6: Average numbers of errors for the students divided by diagram score. The numbers show how big percentage of max possible numbers of errors the students got.

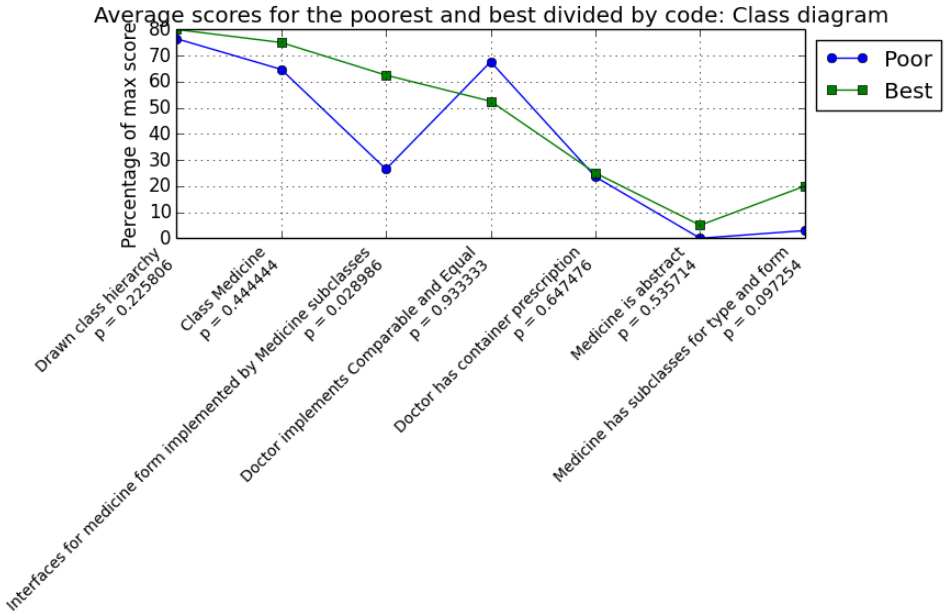


Figure 4.20: Average scores for the class diagram, students grouped by code score.

At the class diagram in figure 4.20 we also find that there is a notable difference when it comes to “Interfaces for medicine form implemented by Medicine subclasses”, but none, in contradiction to the data structure, for the criteria “Medicine is abstract”. In the object diagram we find that there is not any significant differences between the best and the poor. Even though we saw in figure 4.15 and 4.16 that the best deliveries according to the program code also in average is better at diagram, the difference in percentage is not very big.

## 4.4 Comparing poor and best diagrams

In order to research whether there is a difference in the code quality and comprehension of the program at those students that succeed in creating diagrams, we calculate the same numbers for the deliveries divided by their diagram score. We left out the student deliveries not containing *both* diagrams so we could research differences in the program code contra the diagram quality and were left with 39 deliveries.

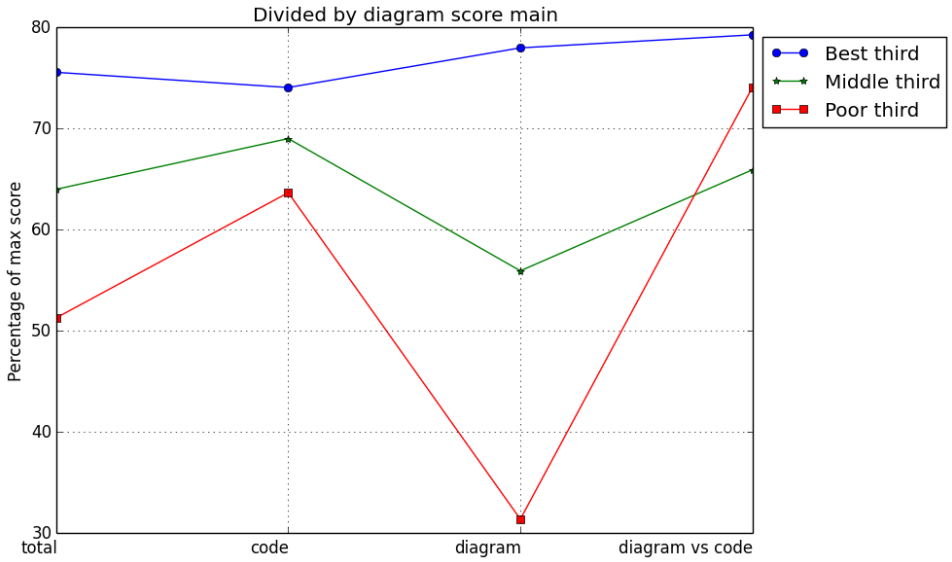


Figure 4.21: Average scores for the best and poorest third divided by diagram score.

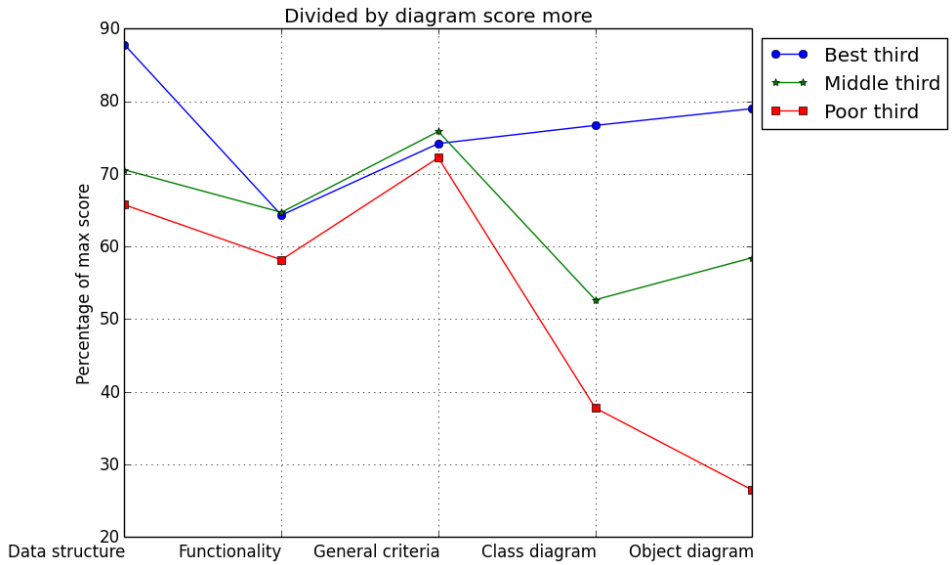


Figure 4.22: Average scores for the best and poorest third divided by diagram score.

There was not much difference between the code criteria when dividing after diagram scores. The average scores on the program code, as shown in figure 4.23, suggests that the best do better at having Medicine as an abstract class. This division also shows a difference when it comes to the subclasses of Medicine and their interfaces. This is also the case for the class diagram in figure 4.24. We see that the poorest on average do better than the best on having a container for prescription in the Doctor class, but this is not close to being significant, with a p-value at 0.62. At

the object diagram in figure 4.25 we see, in contradiction to the previous divisions, some statistically significance between the best and the poorest at OldestFirst list having a pointer to a list of Prescription objects. Even though we see a large difference in the average score, there is not much significant in the object diagrams.

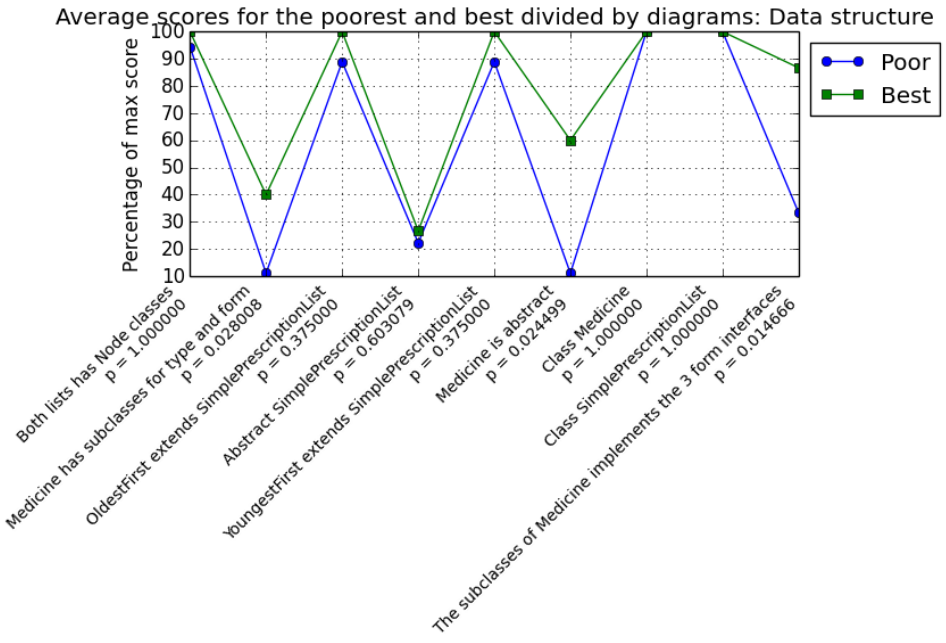


Figure 4.23: Average scores for the data structure, students grouped by diagram score.

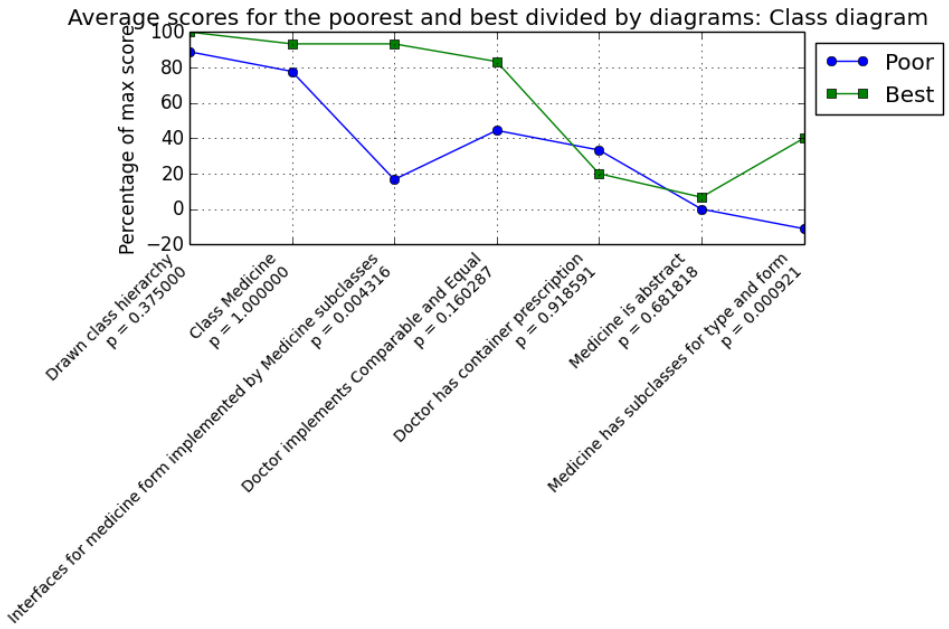


Figure 4.24: Average scores for the class diagram, students grouped by diagram score.

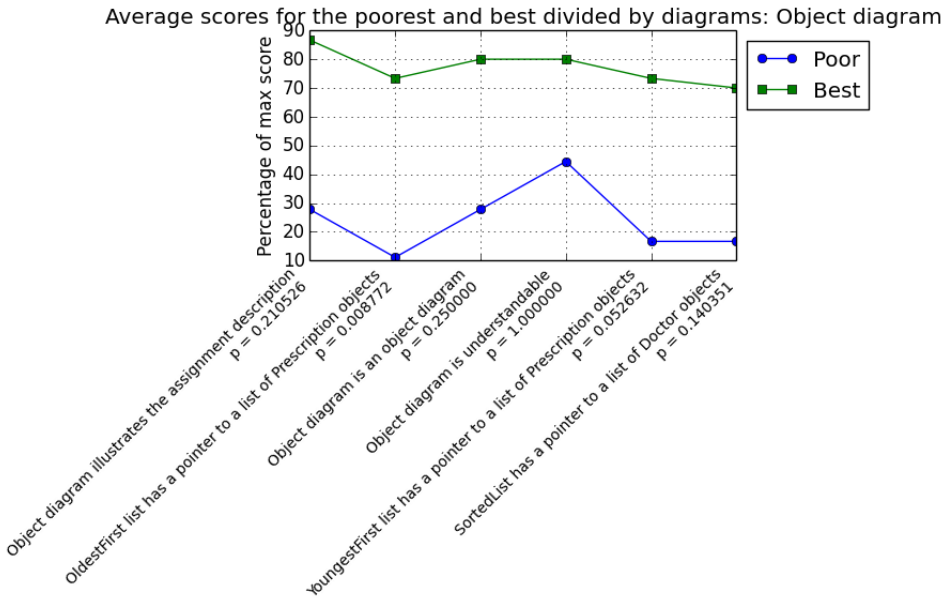


Figure 4.25: Average scores for the object diagram, students grouped by diagram score.

### 4.4.1 Comparing the student groups in the three categories

In table 4.7 the number of student deliveries in the poor, middle and best third are listed for all three division methods described above. The distributions of the deliveries is also marked along the x-axis of the figures 4.1, 4.2 and 4.3. The sizes of the best and poorest thirds were different for each division method, so we found out what deliveries that had been placed differently.

| | **Total** | **Code** | **Diagram** |
|---|---|---|---|
| Poor third | 20 | 17 | 9 |
| Middle third | 19 | 19 | 15 |
| Best third | 17 | 20 | 15 |

Table 4.7: The sizes of the three thirds in the three division methods.

When dividing according to code score, one of the students that were in the total best third actually dropped down to the poorest third while four of the total poor third bumped up to the best third for code.

Three of the deliveries from the poorest third according to the code was to be found in the best third according to the diagrams and two from the poorest group according to diagrams was in the best third according to code.

This is suggesting that the diagrams have an impact on the total score and that those doing well on this but not necessary on code, can score high on total and visa versa.

We did not look into how many that changed up or down from the middle group, since we have not focused much on this group in our analysis, and since it is natural that the ones lying in the lower and upper range of the middle third can bump up or down if we sort the deliveries a bit different.

## 4.5 Diagram compared to the code

After looking at all the deliveries, we chose to look at the correlation between the diagrams and the program code for those that delivered the diagram. The figures in the following subsections describes the criteria that is included in the "diagram versus code" score from all the "main scores" figures 4.4, 4.11, 4.15 and 4.21. The two next subsections will go into detail about the correlation with the code and class diagram and the code and the object diagram.

### 4.5.1 Class hierarchy compared to program code

Figure 4.26 compares how many students that fulfilled the criteria that were both applied to the class hierarchy and the program code. Only the deliveries containing both the class hierarchy and the program code are taken into account (47 in total), since there of course is no match when the diagram is not present.

The y-axis in the figure shows the number of student deliveries that had their written code were according to their diagram. The x-axis shows the relevant criteria. It is worth noting that the origin starts at 24 on the y-axis, showing that all of those that delivered the class diagram had *some* match between diagram and program.

In figure 4.27 the correspondence between code and diagram for the two alternatives for creating the Medicine hierarchy is shown. The alternatives were illustrated in section 3.3. The first two criteria, on the left side of the diagonal line, represents the solution with a total of 12 subclasses. The criteria on the right side represents the solution with 9 medicine subclasses. From the figure, we can see that most student groups had a correspondence between code and diagrams for the first alternative.

The criteria "Medicine has subclasses for type and form" in figure 4.26 is the sum the deliveries that fulfilled *both* of the two criteria on the left side of the line in figure 4.27 and those that only fulfilled the criteria with 9 subclasses. One might notice that the sum of students managing to get the correlation between the Medicine subclasses correct in figure 4.26 is 34, as is the sum of "The subclasses has 3 form subclasses each", while the solution with 9 subclasses adds up to 4 students. This is not wrong, because the two criteria of the solution with 12 subclasses in total shows how many that fulfilled that *exact* criteria in both code and diagrams, but for it to be counted into the "Medicine has subclasses for type and form" from figure 4.26, the student delivery has to also fulfilled the criteria for "3 subclasses of Medicine type".

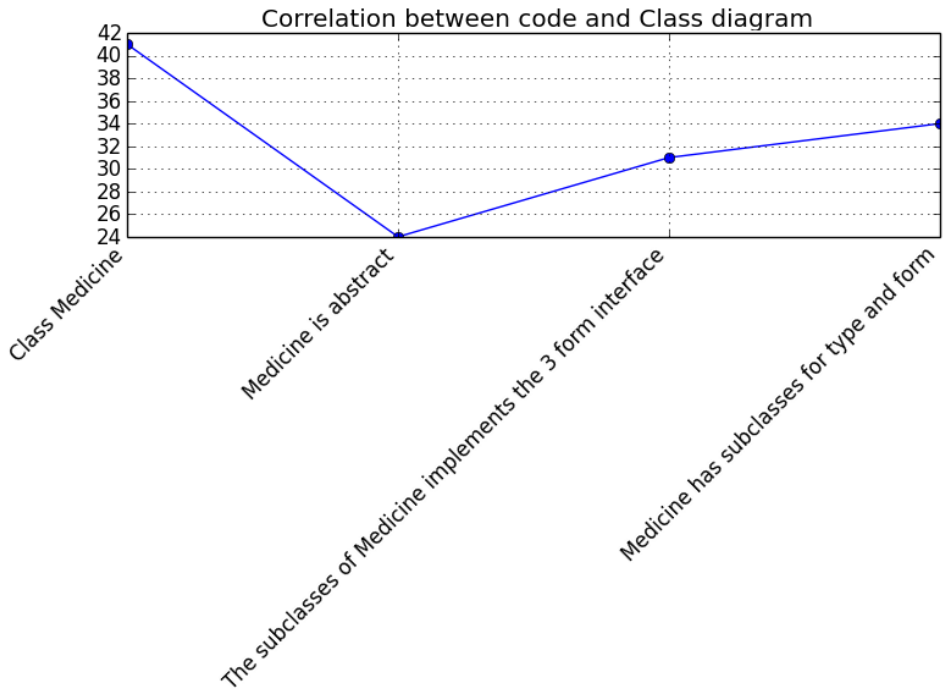


Figure 4.26: Comparing class hierarchy diagram criteria to code criteria

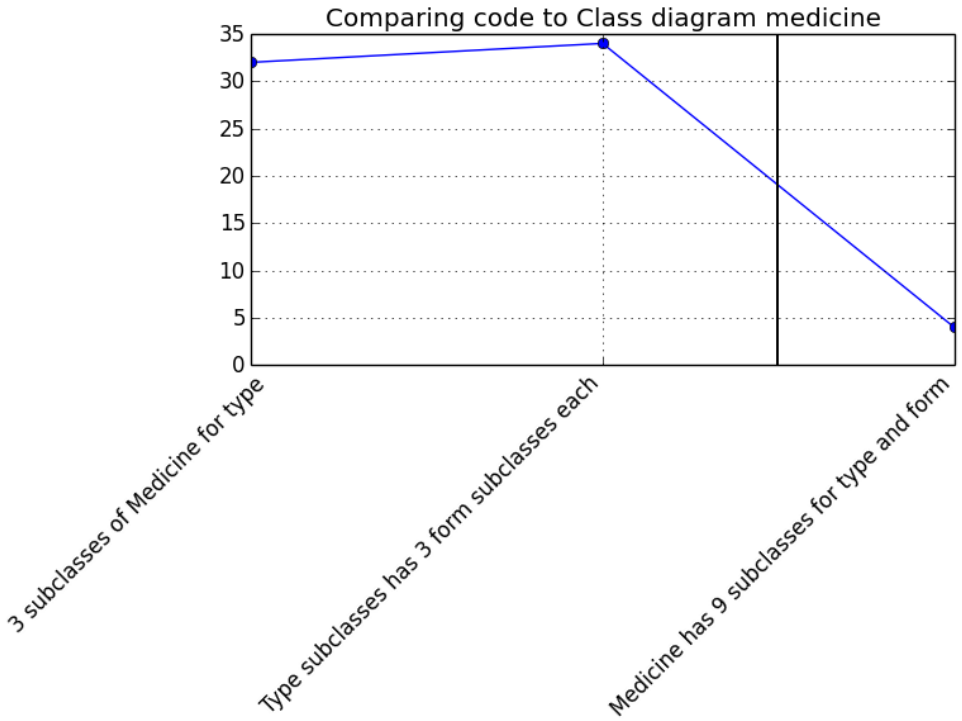


Figure 4.27: Comparing the two alternative solutions for Medicine class hierarchy diagram to code criteria.

### 4.5.2 Object diagram compared to program code

The correlation between the object diagram and the code can be seen in figure 4.28. The 40 deliveries that contained both program code and object diagram is taken into account. Here we can see that the y-axis starts at 0, because only 1 student delivery managed to have a correlation between abstract or not abstract SimpleList. Since the correlation only takes into account that what's drawn in the diagram is the same as programmed, this does not say whether the SimpleList was abstract or not. It is also worth noting that all but 1 of the 40 that delivered object diagram managed to get the correlation between previous pointers in the list nodes correct.

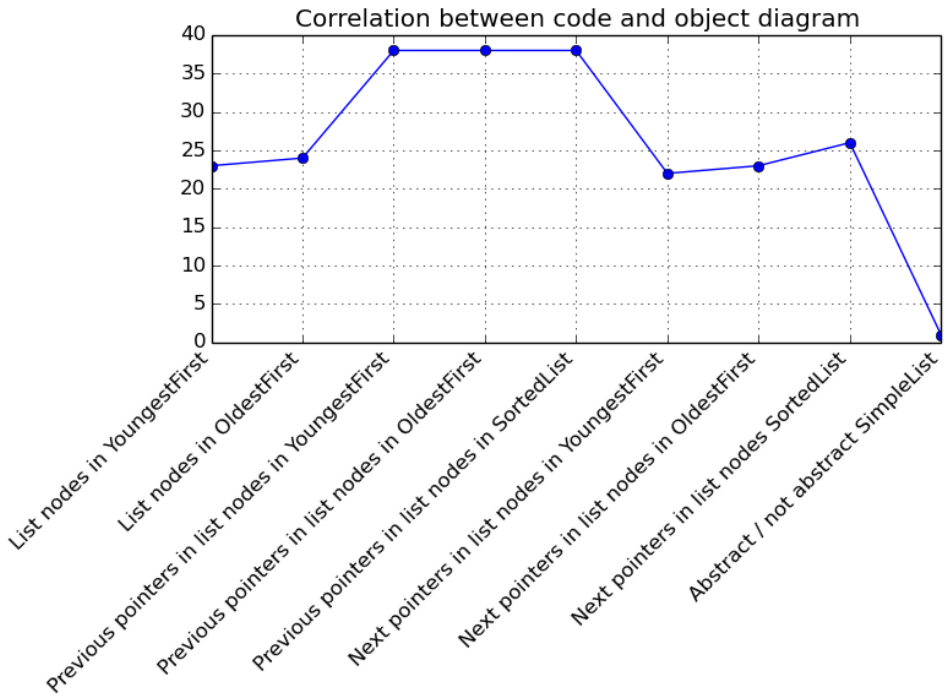


Figure 4.28: Comparing object diagram criteria to code criteria

Since this figure only counts how many student groups that had diagrams and code showing the same, it also counts those groups that did not fulfil a criteria in neither of them. The number of students is high at the previous pointers because most student groups did use previous pointers in the list nodes in code or diagrams. We can see that the next pointers in the list nodes has a lower correspondence, which is caused by most students programming next pointers, but not everyone included them in the object diagrams.

## 4.6 Those that delivered in pairs vs those that delivered alone

As mentioned in section 3.3, some students used pair programming for solving this assignment. 36 of the deliveries were made by pairs, while 22 were made by only one student. From table 4.8 we can see that whether the deliveries were made by one or two students do not seem to make a big difference for the scores. Based on this, we have chosen not to investigate differences related to the size of the groups further.

| | Total score | Code score | Diagram score |
|---|---|---|---|
| Pairs | 63% | 71% | 42% |
| Singles | 64% | 68% | 49% |

Table 4.8: The average scores in percentage of maximum score of the pair deliveries and the single deliveries.

## 4.7 Typical errors

Table 4.9 lists some common errors we found in the deliveries in the data set. 55% of the deliveries contained long methods and code blocks in the menu classes and 27% have this problem generally through the program. There is no official standard of method lengths, but we have marked methods at about 100 lines or more as too long and also concidered the functionality of the methods. A sorted insertion method needs more code lines than a method to return an object variable.

The common errors at the interfaces for the Medicine hierarchy is to be found in table 4.10. We can see that there is not any criteria that stands out, but when it comes to the subclasses 19 students got *some* errors on the hierarchy implementations (and 3 deliveries did not contain anything of the Medicine classes). In the class hierarchy diagram, there was 14 students that had some errors on the hierarchy.

We saw during the division methods, table4.2, 4.4 and 4.6, that the average number of null-pointer errors varied much, the number of errors was at all times under 20% for all the thirds. When divided by code, though, the best thids had only 7% null pointer errors, while the best third had 15% when divided by diagrams core. The poorest third at all division methods had on average about 20% of the total null pointer errors.

### 4.7.1 Abstract classes

All but one of the delivered programmes contained class Medicine and 25 of them had it implemented as an abstract class, but only 2 of the delivered class hierarchy diagrams had marked Medicine as an abstract class. All delivered programs contained the SimpleList while only 14 implemented it abstract and 13 had drawn objects of it in the object diagram. Figure 4.26 and 4.28 shows that the 24 students got the correlation between abstract /

| What | Number of students | Percentage of students |
|---|---|---|
| Uses iterators without checking for null | 19 | 34% |
| Looping through containers using counter variables, not checking for null pointers | 12 | 21% |
| Using node.next without checking for null | 13 | 23% |
| Duplicated code in stead of creating methods for the functionality | 11 | 20% |
| Long methods and code blocks in the menu class(es) | 31 | 55% |
| Long methods and code blocks in general | 15 | 27% |
| Having unused or not necessary / sensible variables | 13 | 23% |

Table 4.9: Common errors of all students. Second column shows the number of students having this error and the third column shows the percentage of students.

| | | |
|---|---|---|
| The interfaces for medicine form is empty | 6 | 11% |
| Implemented one interface for all properties | 8 | 14% |
| The interfaces are implemented by super class Medicine instead of the subclasses | 8 | 14% |
| Class diagram: One interface for all properties | 7 | 13% |
| Class diagram: The interfaces are implemented by super class Medicine instead of the subclasses | 7 | 13% |

Table 4.10: Common errors at the Medicine hierarchy. Second column shows the number of students having this error and the third column shows the percentage of students.

not abstract Medicine class right and 1 got it right for SimpleList. Keep in mind that these figures only show those that delivered the diagrams.

6 deliveries had abstract insertion method in SimpleList and 11 had none. For those that implemented insertion method, 29 made it insert at the end of the list and 15 of those again did not implement any insertion method in the subclass OldestFirst. Strictly speaking, this is not wrong if we look at the assignment description, but since they had an empty subclass, it is not good programming either.

Abstract classes is introduced in the lectures the same day the assignment description is made available to the students and a month before the delivery deadline. They should have had the time to comprehend the concept through the lectures and the small weekly exercises. Still, these numbers suggest that the concept is not understood by many students.

## 4.8 Typical characteristics of the deliveries

In this section we will describe some typical characteristics of a good and a poor delivery for both program and diagrams. We focus on showing examples of the areas found most significant in the earlier sections.

### 4.8.1 Typical diagrams

The delivered diagrams were all over the line from unreadable to exemplary. We found that less students delivered the object diagrams than the class hierarchy and that the information we wanted to extract from the object diagram was hard to find. The detail level was often to low and many of the diagrams were hard to interpret because of e.g. bad handwriting.

We now present two destilled examples of the poorer diagram deliveries. Note that we have chosen not to include the unreadable examples, as this did not give us much information about student comprehension. In addition to the two presented here, there was also several diagrams that did not add a pointer from the doctor and person objects to their prescription containers, OldestFirstList and YoungestFirstList respectively.

#### Class diagram errors

We observed that the main reason for not scoring high on the class diagrams was that students seemingly had interpreted the program description wrong, thus creating diagrams not representing everything we had as criteria. The main criteria missed was wrong amount of medicine subclasses and not implemeting interfaces as well as not having marked medicine as an abstract class.

#### Lists without head nodes

Most of the object diagrams showed the lists without a list head node and many illustrated a list structure without nodes. We also noted that many

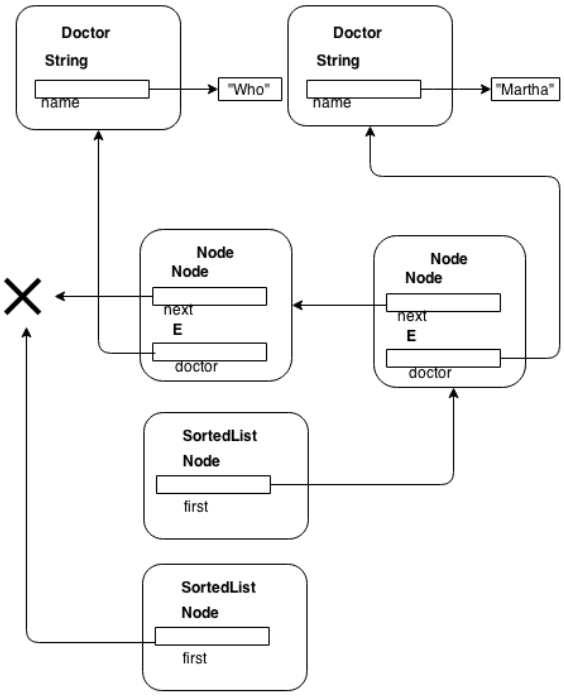


Figure 4.29: Example of how the objects diagrams represented lists without list heads.

did not draw a pointer from the list object to the objects inserted in the list. Figure 4.29 shows an example of how many illustrated lists without using list head nodes. One list is containing objects while the other is empty, thus first is pointing directly at null. The alternative using list nodes is illustrated in section 3.3. Note that some students skipped the Node-objects, just drawing arrows between the Doctor objects.

#### Representing lists as arrays

Another mistake we found interesting in the object diagrams was that the lists was represented as arrays and not drawing next pointers between the objects. 25% of the delivered object diagrams did this in stead of the more usual list structure. An example is shown in figure 4.30.

### 4.8.2 Typical programs

We will also show some examples of the typical parts of the code we found interesting.

#### The medicine hierarchy

First an example of a good implementation of the medicine class hierarchy, shown in Figure 4.31.

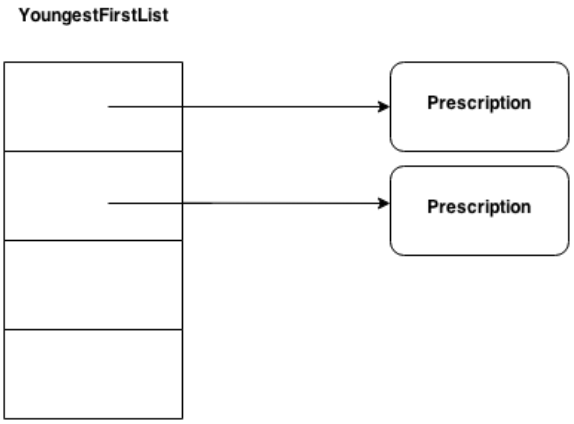


Figure 4.30: Example of how some object diagrams illustrated the lists as arrays.

```
abstract class Medicine {}

abstract class TypeA extends Medicine {}
abstract class TypeB extends Medicine {}
abstract class TypeC extends Medicine {}

class TypeAPill extends TypeA implements Pill {}
class TypeALiniment extends TypeA implements Liniment {}
class TypeAInjection extends TypeA implements Injection {}

class TypeBPill extends TypeB implements Pill {}
... \\ and so on for rest of the interfaces and for TypeC
```

Figure 4.31: Example of a good implementation of the Medicine hierarchy. Many of the good implementations had done everything like this exept for having the TypeA, TypeB and TypeC as abstract.

```
class Medicine implements Type {}

class Pill extends Medicine {}

class Liniment extends Medicine {}

class Injection extends Medicine {}
```

Figure 4.32: Example showing the class signatures of one of the most commonly found wrongful implementations of the Medicine class hierarchy. The interface should be implemented as subclasses and the subclasses should have been interfaces and the interface should not be implemented by the super class. Also, this implementation is missing the subclasses of the subclasses.

```
class Medicine implements Type, Form {}
```

Figure 4.33: Example of a poorer implementation, with only one class in the hierarchy. This example shows the class implementing interfaces for medicine form and type, but some of the poorer did not implement any interfaces.

Figure 4.32 and 4.33 show two of the most common wrongful implementations of the medicine hierarchy. The deliveries not managing to implement it after the assignment spesifications mostly got the number of subclasses wrong, and thus not implementing the interfaces at the correct classes, or did not create interfaces for medicine form.

### List nodes and next-references

The students not using Node-objects to create the lists created a list of the objects to be stored, declaring next-references in the Prescription or Doctor classes instead. Not all managed to actually create lists in the list classes. Some did not implement these classes, but those that did implemented container functionality by declaring arrays for storing the objects.

### Method division

As seen in table 4.9 of the most common errors was to have too long methods and code blocks. This was particularly bad for the menu and user interface part and since the assignment text is not as detailed about how this should be implemented as for the rest of the program, this is perhaps not that surprising. Less expected was to find that this was a quite common problem for other parts of the program, where typical the insertion methods for the lists were too long (above 100 lines of code). We saw that not only were the methods very long, but they could easily have

moved several tasks into seperate methods. A good example of this is when the students use nested loops to do several things, each loop could be moved to a new method.

Figure 4.34 shows an example of a method inserting sorted in a list. It is not a very long method, but it repeats for all the if-else tests these three statements `x.next; size++; return;`. The example is inspired by some of the deliveries creating longer methods for insertion, but it is not copied from any of them, just a recap of what they all had in common. We have renamed method and variable names, not included any comments, so this is not traceable back to any student deliveries.

```
public void insert(E e){
    Node n = new Node(e);
    Node tmp = first;

    if (first.next == null){
        first.next = n;
        size++;
        return;
    }

    if(e.compareTo(tmp.next.e) <= 0){
        n.next = tmp.next;
        tmp.next = n;
        size++;
        return;
    }

    tmp = first.next;
    while(tmp != null){
        if(tmp.next != null){
            if(e.compareTo(tmp.next.e) <= 0){
                n.next = tmp.next;
                tmp.next = n;
                size++;
                return;
            }
        }else{
            tmp.next = n;
            size++;
            return;
        }
        tmp = tmp.next;
    }
}
```

Figure 4.34: Example of insertion method with lots of repetive code.

As we can see the repetitive statements is adding to the length of the method and keep in mind that SOLID principles states that a method should have *one and only one* function. A shorter version, where the actual insertion is happening in another method is to be found in figure 4.35. This example is inspired by some deliveries containing good solutions for list insertion, but as for the previous example, it is rewritten and not a direct copy.

```
// Finds the correct place in the list to insert element e
void insert(E e) {
  if (isEmpty()) {
    insertAfter(e, null);
    return;
  }

  Node tmp = head;
  while (tmp.next != tail) {
    if (tmp.e.compareTo(e) > 0)
      break;
    }
    tmp = tmp.next;
  }
  insertAfter(e, tmp);
}

void insertAfter(E e, Node after) {
  Node node = new Node(e);
  size++;
  if (after == null) {
    head.next = node;
    return;
  }

  node.next = after.next;
  after.next = node;
}
```

Figure 4.35: Example of a shorter insertion method.

### List tails and null-cheking

Figure 4.34 also shows an example of how students found end of the list without using a list tail. For the sorted list, it was not needed but the assignment description requires the students to declare a list tail in the SimpleList. Even though it is not say anything about using it, it should be implicitly that a declared variable should be used to something. A typical example of list traversial using counter variables, not checking for null in

the list is shown in figure 4.36.

```
E search(K key) {
  Node tmp = head;
  for (int i = 0; i < listSize; i++) {
    if (key.compareTo(tmp.key) == 0) {
      return tmp.e;
    }
    tmp = tmp.next;
  }
  return null;
}
```

Figure 4.36: Example code that is using counter variables for list traversial ant not checking for null.

## 4.9 Summary

We have seen that there is not many significantly differences between the average of the best and the poor, but for some criteria we could say that there is evidence against our null hypothesis, meaning that there is a statistically significance at our chosen level 0.05 that the average best deliveries is better than the average poor at:

1. *When divided by total score:*
   (a) Medicine has subclasses for type and form - *Both in code and class diagram*
   (b) The interfaces for medicine form implemented by Medicine subclasses - *Both in code and class diagram*
2. *When divided by code:*
   (a) Medicine has subclasses for type and form - *In code*
   (b) The interfaces for medicine form implemented by Medicine subclasses - *Both in code and class diagram*
   (c) Medicine is abstract - *In code*
   (d) Search in SimpleList checks for null
   (e) Insertion in SortedList uses compareTo
   (f) Division of tasks into reasonable methods
3. *When divided by diagrams:*
   (a) Medicine has subclasses for type and form - *Both in code and class diagram*
   (b) The interfaces for medicine form implemented by Medicine subclasses - *Both in code and class diagram*

(c) Medicine is abstract - *In code*

(d) OldestFirst list has a pointer to a list of Prescription objects - *In code*

Note that all the division methods suggests that the best deliveries is better at the interfaces for the Medicine subclasses at both code and class diagram and at implementing the correct subclasses of Medicine. Also worth noting is that the best is better at programming Medicine as an abstract class when divided by code and by diagrams, but not when divided by total score.

The students with the highest score on the code is not necessary the same that have high score on diagrams. The correlation scores is in general quite high, when we only look at those that delivered diagrams, the average for best is almost 80%, for the middle 65% and for the poorest deliveries 75%. Most of the students implemented insertion method in SimpleList. The correlation test between code score and diagram score strongly suggests that there is a significant correlation between high code score and high diagram score.

In the next chapter we will discuss whether we can generalise these results and the consequences of them.

# Chapter 5

# Discussion

In this chapter we will discuss the results presented in the previous, their validity and whether we can generalise them. We also present the possible reasons for the results we found and whether there are external factors that can have affected our findings. Then we will discuss the focus in the university course and the assignment description and how this can affect the student groups diagrams, before we compare our results with other research.

## 5.1 The findings from the analyse

We will start by discussing the results from our analyse of the categorised student groups, how the chosen criteria and their weights can have an impact on our findings and how we have interpreted the delivered diagrams.

### 5.1.1 Comparing the categories

As we compared the student groups categorised as poorest and best, we found that there was not many significant differences between them. This could be because there were little differences between the deliveries and most students did well and struggled on the same things. It could also have been affected by how we analysed them, as we will discuss in the next sections.

### 5.1.2 The criteria we chose and how we weighted them

As described in section 3.5, we focused on parts of a bigger assignment, meaning that if we have chosen different aspects our results could have been different. An example is that we did not look into the generic container Table, which was the first containers to implement according to the sequence in the assignment description. Therefore, and since arrays are a well known concept to the students, it is possible that we would have had a higher percentage that fulfilled criteria for the Table container than we did for SimpleList and SortedList.

We wanted to see how the students compared the Doctor objects in the SortedList, so we chose to focus on the implementation and use of the Java built-in interface Comparable. We could have looked at the interface Equal instead, which they were supposed to write them selves, thus the students might have given this interface and its applications more thought. Still we observed that most students managed both to implement Comparable in the doctor class and use its `compareTo`-method when comparing the objects. Looking at the interface Equal could thus result in a lower number of students scoring on the criteria related to this, giving us more differences between the student groups to look into.

#### The point system

We have weighted the criteria as worth 1 or 2 points, thus making some criteria count more than others. The weights were based on how difficult we assumed that exact criteria to be, which again to some degree had root in the amount of code it required, as explained in more detail in section 3.5. The evaluations of how difficult a criteria is could have affected our results. Perhaps should criteria like having classes as abstract be worth 2 points instead of 1, based on our observation that there were few that managed this.

### 5.1.3 Diagrams

When categorising the deliveries, we wanted to see whether the student groups categorised as the best groups according to code quality had better diagram quality than the poor groups. We also wanted to see whether it was a correspondence between the best student groups according to diagram score and high score on the code criteria. Neither of these were the case.

We saw in chapter 4 that for all student groups, the delivered diagrams were poorer than the corresponding program code with average scores on 45% and 70%, respectively. These results include those that did not deliver diagrams, but when excluding those that did not deliver any diagrams, the average diagram score jumps up to 52%, meaning that the average code is better than the average diagrams.

The best student groups according to the diagram score did slightly better at object diagrams than on the class diagram, as goes for the middle student groups. The poorest groups did about 10% worse on object diagram than the class diagram. When looking at the code score, the student groups in the three categories on average did worse on the object diagram than the class diagram and we see that there is not much difference between the three categories, suggesting that most students did not do well on the object diagrams. This is suggesting that the drawing of diagrams in this student assignment did not create any significant difference for the student groups code quality.

#### Our interpretation of the diagrams

We did not focus much on the notation in the diagrams, whether it was following the university course's notation, standardised UML or undefined notation, as long as we understood what it was trying to illustrate. Still, it is possible that we have interpreted some diagrams wrong, giving too much or too little points for some criteria. Also possible is that we more easily gave points to diagrams that were legibly than those with bad handwriting or contained many arrows criss-crossing, making them hard to follow.

### 5.1.4 Diagram errors

Several student groups illustrated the list in the object diagram as an array, as shown in figure 4.30. While a few of these actually had implemented the list classes with an internal array in stead of creating a list, most students had programmed a normal list structure with list objects having next-pointers, resulting in the lists being represented different in diagram and code. It is not possible for us to know whether the reason for this is that the students have misunderstood the list structure as a concept, just did not think through the data structure when they created the diagram or did not know how to represent the lists in a diagram.

Another repetitive error in the object diagrams was that some objects where not referenced from anywhere, e.g. the OldestFirst lists where not referenced from any Doctor object, thus "lost in time and space". There is possible that the student groups did not think this were relevant information, therefore not including it in the diagrams.

### 5.1.5 Program errors

The analyse showed that there were several errors present in multiple programs. This section will discuss the most common errors and possible reasons for them being common.

#### Counter variables for list traversal

We saw that about 1/5 of the students used counter variables in list traversal, trusting the list to be correct and thus not checking for null pointers. A linked list example from a lecture used counter variables for traversal without checking for null. Even though the paper about linked list that is on the curriculum is traversing the list using the next pointers in the nodes, there is a possibility that the students find both solution equal, choosing to program the one they recognise from array traversal. [1]

#### List nodes versus next references in the objects

The first examples of lists in the lectures does not use Node objects for creating the list structure. Instead the lists have a reference of that type

[1] Example from lecture of linked list using counter variables for traversing the list: http://heim.ifi.uio.no/ inf1010/v14/lysark/LenkeListe.java

that is to be stored in the list (first example is a Dictionary list holding Word objects) and the objects to be stored have a reference to the next object in the list. When they a short time later introduce Node objects for containing the objects and reference to next node in the list, there is possible that some students do not understand the point of the difference between the two ways of creating the list. The way the lectures teach both the "simple way" and the more standard way could make the students thinking there are equally good solutions, thus choosing to create their own lists in the simplest way.

#### Null pointers

When students e.g. are implementing a list, it seems that many takes for granted that they implemented the list fault proof, so that there are no null-pointers in places they do not check for null. This can also be a reason for why there were so many that used counter variables without checking for null.

### 5.1.6 Code principles

There is not much focus on code principles in the curriculum. This, in addition to the size of the programs to be written, can cause the students to focus on *creating a program that works* rather than how its implemented.

Dividing code into reasonable methods, following code principles, probably requires a good comprehension of the program and its tasks. The strict program description might leave the students to follow it without thinking about good and poor implementations, not encouraging creative thinking. This could result in that some students is scoring less points at our general criteria, since some of these requires the code to be implemented iin a good way.

## 5.2 The focus of the course

As explained in section 2.3, the lecturers and curriculum have almost no focus on how to illustrate the functionality of a program or how to visualise it before writing the code. The quality of diagrams can suffer from the lack of teaching how to create them because it requires understanding not only of the program structure and functionality but the of the drawing technique as well.

The object diagram the students are requested to create in the researched assignment shall illustrate the elements inserted in the containers and that some objects exists in more than one container. Some fields in the objects should be included, like next pointers in the list nodes and pointers from objects to their lists. Other fields are not relevant, like a persons age or a doctors specialisation. It is not the easiest object diagram to create when the students have little experience and not a curriculum description to walk them through it.

The class diagrams are more simple, as they should not include any content of the classes, just what their signature says. Still we found that the score on these diagrams on average were almost the same as for the object diagrams, but this seems to be caused by the wide spread on the point axis, as indicated in figure 4.22. The best student groups, according to diagram score, delivered much better object diagrams than the poorer groups.

### 5.2.1 The assignment description and the diagrams

The description of the assignment the students are given is very detailed, describing all the necessary classes, their functionality and how they are connected. This does not leave much interpretation or freedom to the students and those with good comprehension of programming and Java could probably get good enough overview of the program by reading the description. Therefore it is possible that some student groups did not need to draw the system in order to understand it, thus did not put much work into these diagrams or perhaps did not deliver them at all. This could have an effect on our results by creating a correlation between high code score and low diagram score.

There is also possible that some groups delivered the diagram to their teaching assistant by hand or that the data set we have examined did not contain all the diagrams uploaded to the delivery system. Recall from section 3.4 that there was possible to upload files to the system several times before the deadline and that our data set only consisted of the last deliveries.

According to the assignment description the diagrams shall be created before programming the corresponding code. As mentioned in chapter3, we have witnessed many students solving the assignment, and we observed that several groups created the diagrams after programming the assignment. Some groups started drawing the class diagram before programming, but stopped before completing it explaining that they had understood the class hierarchy so they rather wanted to focus on the programming.

If the thought behind making the students creating diagrams before programming is for them to have a clear idea of what to program, it does not seem that the intention is communicated clearly to the students.

Still, if the students created the diagrams after they have coded the program, the diagrams works more like an explanation or documentation of the actual program rather than a guide to how to create it. If this is the case, its content should anyway correspond with the content of the program. We found that the correspondence between diagrams and code were quite high, even when the diagram quality were poor.

With the pair programming development technique introduced for the first time in the course, perhaps the drawing as a development technique drowns in the mess. We saw that there were little differences in the average scores of the groups that consisted of two students and those that consisted of one. There is a possibility that the pair programming process did not do any significant difference for the delivered programs and diagrams.

### 5.2.2 Diagrams as a step in a development technique

We set out to research whether creating diagrams would help the students as a part of the development and problem solving process. Our results did not suggest that it helped and we observed in the class rooms that whether the student groups drew and completed the diagrams in front of programming varied a lot. Our impression is that some students fail to see a connection between the diagram and the implementation, instead looking at them in separate and are not thinking about what the diagrams illustrate when they are programming. Also, we observed that some students seemingly did not feel the need to create diagrams, thus only doing it because it were a mandatory part of the assignment. On the other hand, we observed that some groups of students used the time drawing diagrams to discuss, explain and brainstorming the program with their group partners.

## 5.3 Comparison with other research

Since our data set were a random sample of student groups, we assume it to be representative for the course.

Our results suggests that there was no significant difference on the diagrams of those that did well on writing code compared to those that did poorly. This is not according to the results of Holliday and Luginbuhl, which found a correlation between the students ability to construct object diagrams and the students comprehension of object-oriented concepts (Holliday and Luginbuhl, 2004). Even thought their memory diagrams is comparable to the object diagrams in our thesis, the setting was different. They mainly observed students in the classroom and both the diagrams and code fragments were smaller, while our study case were a larger system and the data set we analysed is not possible to connect to the students we observed in the classroom. Our findings is supporting the study of Thomas, Ratcliffe and Thomasson more, which concluded that the diagrams did not help the students in answering multiple-coice programming questions. Research on diagrams and illustrations to increase comprehension when studying electrical components suggests that illustrations and diagrams were usefull (Mayer, 1989). Using diagrams to create something, e.g. implement a program, is not the same thing as understanding how something is built, which may be one reason for our research not supporting Mayer's results.

The assignment description did not seem to leave the students in a creative and freely thinking state for solving the problem and implementing the solution. There is possible that if the students were given an assignment where the classes and their content was not that strictly listed, they would made more use of creating diagrams, e.g. as an analysation and planning technique, rather than seeing it as an obligation.

Our study did not show a correlation between code quality and diagrams quality. One reason for this could be that the students do not see creating

diagrams as a way of planning a solution strategy for the problem. An other reason could be that students do not use it as a tool for ensuring that they are writing the program as they intended it to be, resulting in that the students get stuck in their implementations, even though they might not follow their original plan.

# Chapter 6

# Conclusions and future work

In this thesis we wanted to research first year informatics students comprehension of programs compared to diagrams describing the program. We will now recap the answers to our research questions from section 1.3 and 3.2.4 and, based on our research, suggest some improvements to the course before presenting some future work.

## 6.1 Conclusions

This section lists out research questions and their conclusions. First, we answer the questions related to the categorisation of the student deliveries before, with these answers in mind, moving on to the main questions for our research.

### 6.1.1 Specific research questions

We listed some specific research questions, as goals for the categorisation of the student groups, and we will now list the questions we set out to answer.

**What do the deliveries have in common? What do the poorer deliveries have in common and what are the similarities between the good deliveries?** We saw that all deliveries struggled with the length of methods and code blocks in the menu and user interface. Poor deliveries struggled with abstract classes and the Medicine hierarchy, had more null pointer errors than the good deliveries.

**Is there a connection between the program (the actual code) and the diagrams? Are there good code without good diagrams? Is there good diagrams with bad code?** We did not find a connection between the program and the diagrams for the student deliveries. Some student groups delivered good code with bad diagrams, while some delivered good diagrams with bad code. One reason for this can be that

students with good comprehension do not see any value in drawing the diagrams, resulting in them not putting much effort in creating them. An explanation for poor students creating good diagrams can be that they have studied the technique from the lecture notes or that they think it is easier to create diagrams for a description of a program than writing the code. In that case, creating diagrams does not seem to help the students comprehend the concepts or in writing better code.

**What is missing to elevate the poorer students up to an average or good level?** Only looking at the code, we found that the poorer students struggles with the class hierarchies and interfaces, length of the methods and null pointer errors. To lift them to a higher level, the students should be allowed to focus more on these topics by programming smaller tasks, such as creating smaller class hierarchies. That way they could comprehend the concept without learning others in parallel, which they in the researched assignment have to do. It could also help if the course would focus more on how to write good code and show the students the difference between debugging long code blocks and methods compared to smaller, one-task-only methods. By learning to break the problem into smaller pieces and tasks there can be easier to solve bigger assignments.

**Can we state something about the students understanding and miscomprehension by comparing code and diagrams?** We did not find any significant retaliation between good code and good diagrams or poor code and poor diagrams. There were good code with both bad and poor diagrams as well as poor code with both good and poor diagrams.

### 6.1.2 Research questions and conclusions

We wanted our comparisons of diagrams and programs to give us some indications about the students comprehensions and misunderstanding of programming. What we found did not indicate that the diagrams helped the students a lot, but we still saw that in many cases the correspondence between the code and diagrams were high, meaning that the errors we found in diagrams often were present in the code as well.

**Will visualising the program ahead of programming it help the students in writing better code and to comprehend the concepts at hand?** We did not find that there were an significant difference between the diagrams of those that did well and those that did poorly on the code. We found some minor differences the poorer students struggled significantly more than the best with, like subclasses and interfaces, abstract classes and division of tasks into reasonable methods. These did not make significant difference looking at the overall picture. Neither did we find enough evidence to prove that creating diagrams of the program structure indicated any difference between the quality of the code. This may be due to how the course is set up, not focusing on how or why to

create diagrams. If we had done systematic investigations with teaching of diagrams, our results could have been different.

At least, in order to make use of drawing diagrams as a way to write better programs, one need to make them more detailed that the ones the students made.

**Is there a connection between good code and good program diagrams?** We did not find that there were any significant relation between quality of the code and the diagrams. If our perception about students do not seeing the value in creating diagrams is correct, this could make an impact on our findings in this thesis. To clearly communicate the intentions and use of diagrams could result in the good students actually creating diagrams and the poor students increasing understanding of the code. Especially those that created good diagrams but bad code could benefit from knowing how to connect the drawn diagram with the program code by e.g. using the diagram as a guide to code according to.

## 6.2 Suggestions for improvement of teaching

One suggestion for improving the university course is that it need to teach the students how to create diagrams and clearly communicate intentions to do it.

Our opinion is that the course is ambiguous about whether the diagrams are a part of the curriculum or not and should either make it clearer that it is, or leave it completely out of assignments, curriculum and exams. It could also make it easier for the students if they did not have to do two quite new things (creating diagrams and testing pair programming) at the same time.

If the course could focus more on the little tasks, and move away from the large programs that is doing *everything*, perhaps the students would focus more on understanding the concepts.

## 6.3 Future work

We recommend that the course will run a pilot project on giving lectures on how to create diagrams and visualising the programming concepts and what the advantages by doing it, there would be interesting to do an analysis, in a similar way as in this thesis, of how students are interpreting their programmes compared to the actual program code.

Another interesting research would be to analyse the use of diagrams to plan the program structure, e.g. done by observing two classes where one is taught to create diagrams and why while the other is not. The analyse could then be done by giving programming assignments to the students in the two classes, comparing how well the students in the two classes do and if they create diagrams as a part of solving the problems. If this could be arranged as a part of a programming course one could compare the program quality

of the students in the two classes knowing that one group is taught the intentions and techniques.

Based on our observation that some student groups used and drew diagrams as a part of discussing the program before and while implementing it, it we think it also would be interesting to study the use of diagrams as a communication tool in pair-programming. Studying the diagrams and program code of student groups that uses diagrams when communicating can give a more detailed insight in how lesser experienced programming students comprehend programming concepts and diagrams.

Last, we would find it interesting to do a larger qualitative study by interviewing several students while they create diagrams illustrating a solution to a simpler problem and then programming the code. This could give a deeper insight in how students think while the analyse a problem, find solution strategies and whether the diagrams help them developing the solution. Our research can not conclude about the students opinions about creating diagrams and it would be interesting to know know, in a study like this, why the students create them.

# Bibliography


Astrachan, Owen (1998). “Concrete Teaching: Hooks and Props As Instructional Technology”. In: *SIGCSE Bull.* 30.3, pp. 21–24.

Beck, Kent and Fowler Martin (1999). “Bad Smells in Code”. In: *Refactoring: improving the design of existing code*. Pearson Education India, pp. 63–72.

Börstler, Jürgen et al. (2008). “Transitioning to OOP/Java—A Never Ending Story”. In: *Reflections on the Teaching of Programming*. Springer, pp. 80–97.

Curzon, Paul (2002). “Computing Without Computers - A Gentle Introduction to Computer Programming, Data Structures and Algorithms”. Version 0.13. URL: http://www.eecs.qmul.ac.uk/~pc/research/education/puzzles/reading/.

Department of Informatics (2015). *Object oriented programming - University of Oslo - University of Oslo:* [Online; accessed 6-January-2015]. URL: http://www.uio.no/studier/emner/matnat/ifi/INF1010/index-eng.html.

Drake, Peter (2006). *Data Structures and Algorithms in Java*. Pearson/-Prentice Hall.

Gjessing, Stein (2012). *Litt om datastrukturer i Java*. [Online; accessed 6-January-2015]. URL: http://www.uio.no/studier/emner/matnat/ifi/INF1010/v12/forelesningsnotater/OmDataStrukt-2012.pdf.

Harvard College (2015). *This is CS50*. [Online; accessed 13-February-2015. URL: https://cs50.harvard.edu.

Holliday, Mark A. and David Luginbuhl (2004). “CS1 Assessment Using Memory Diagrams”. In: *SIGCSE Bull.* 36.1, pp. 200–204.

Lingjærde, Ole Christian et al. (2011). *Rett På Java*. 3rd ed. Oslo: Universitetsforl.

Malan, David J. (2010). “Reinventing CS50”. In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE ’10. ACM, pp. 152–156.

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Prentice Hall PTR.

Martin, Robert Cecil (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR.

Mayer, Richard E (1989). “Models for understanding”. In: vol. 59. 1. Sage Publications, pp. 43–64.

Miles, Russ and Kim Hamilton (2006). *Learning UML 2.0*. O'Reilly Media, Inc.

Pilone, Dan and Neil Pitman (2005). *UML 2.0 in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc.

Raymond, Eric S. (2003). *The Art of UNIX Programming*. Pearson Education.

Rist, Robert S. (2004). "Learning to Program: Schema Creation, Application, and Evaluation". In: *Computer Science Education Research*. CRC Press, pp. 175–195.

Schoenfeld, Alan H (1992). "Learning to Think Mathematically: Problem Solving, Metacognition, and Sense-Making in Mathematics". In: *Handbook for Research on Mathematics Teaching and Learning*. Ed. by D Grouws. MacMillan, pp. 334–370.

Storleer, Stein Michael (2013). *Lenkelister og beholdere av lenkelister*. [Online; accessed 6-January-2015]. URL: `http://heim.ifi.uio.no/inf1010/v13/notater/lenkelister.pdf`.

Thomas, Lynda, Mark Ratcliffe and Benjy Thomasson (2004). "Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results". In: vol. 36. 1. ACM, pp. 250–254.

UC Berkeley EECS Department (2015). *CS10 The Beauty & Joy of Computing*. [Online; accessed 13-February-2015]. URL: `http://cs10.org/sp15/`.

Woods, John F. (1991). *Usage of comma operator*. [Online; accessed 6-February-2015]. URL: `https://groups.google.com/forum/#!msg/comp.lang.c++/rYCO5yn4lXw/oITtSkZOtoUJ`.

# Appendices

## A: The assignment description

Obligatorisk oppgave 4 i INF1010, våren 2014: "Leger og resepter" Versjon 1.1

*Denne oppgaven skal løses to og to vha. systemutviklingsmetoden "Parprogrammering". For å få levere må* ***alle*** *registrere seg gjennom et enkelt skjema* ***innen 21. februar****. Selve innleveringen skal bestå av Java-program og utfylt skjema fra parprogrammeringen. Krav og praktisk informasjon ligger på filen som ligger her.*

Det er meningen at du skal jobbe med denne obligatoriske oppgaven i fire uker fra 19. februar til 18. mars. Mye av pensumstoffet du trenger i denne oppgaven er allerede gjennomgått. Det siste (om Iterator og Comparable) blir gjennomgått 26. februar.

Du skal i denne oppgaven ikke bruke beholder-klasser fra Java biblioteket (ikke ArrayList, HashMap, etc.), men skrive alle beholderne selv ved hjelp av tabeller (array) eller lister.

Oppgaven skal besvares sekvensielt, dvs. løs først oppgave 1, så oppgave 2, osv. Likevel er det lurt å lese hele oppgaven før du starter å besvare den. Spesielt er det lurt å se på oppgave 7, der du skal lage et program som bruker alle delene som er beskrevet tidligere.

Du skal i denne oppgavene implementere grensesnittet Iterator minst to steder. Hvordan dette gjøres vil bli nøye gjennomgått på gruppene.

Om du ikke er vant til å lage et ordrestyrt program (se oppgave 7), kan du følge en av ekstragruppene, der dette vil bli gjennomgått.

## 1. KLASSEHIERARKIET

### Legemidler

Et legemiddel har et navn, et unikt nummer og en pris. Når nye legemidler registreres gis de et nytt løpende (unikt) nummer som starter på null.
Et legemiddel er enten av type A, narkotisk, eller av type B, vanedannende, eller av type C, vanlige legemidler.

Legemidler av type A har i tillegg et heltall som sier hvor sterkt narkotisk det er.
Legemidler av type B har i tillegg et heltall som sier hvor vanedannende det er.
Legemidler av type C har ingen nye egenskaper (annet enn klassen).

I tillegg til egenskapene beskrevet over kommer legemidler enten som piller, som liniment (salve) eller som injeksjon. Disse egenskapene skal du beskrive vha. grensesnitt.

For enkelhets skyld skal du for piller bare lagre hvor mange piller det er i en eske. For liniment skal det beskrives hvor mange cm3 det er i en tube. For injeksjonsjoner skal det beskrives hvor mye virkemiddel det er i en dose (i mg).

Resepter

En resept har et unikt nummer som starter på null med første resept som opprettes. En resept inneholder en peker til et legemiddel, en peker til den legen som har skrevet ut resepten, og nummeret til den personen som eier resepten (se nedenfor om leger og personer).
En resept har et antall ganger som er igjen på resepten (kalles "reit"). Hvis antall ganger igjen er null, er resepten ugyldig.

Noen resepter er blå, andre er hvite. Blå resepter er sterkt subsidiert, og for enkelhets skyld sier vi her at de er gratis.

Leger

En lege har et unikt navn. Legene skal kunne sorteres alfabetisk etter navn, og man skal kunne finne en lege basert på navn. Klassen Lege skal derfor implementere grensesnittene Comparable (med seg selv) og Lik.

Grensesnittet Lik inneholder en metode kalt "samme" som har som parameter en String og returnerer sann eller usann. Dette grensesnittet kan f.eks. brukes til å finne om et objekt som inneholder et navn (String) har samme navn som parameteren til metoden.

En lege er enten en spesialist eller så er vedkommende ikke spesialist (vanlig lege).

Noen leger har avtaler med kommunen der de jobber (avtaleleger). For en avtalelege finnes det et avtalenummer. Dette er en egenskap både vanlige leger og spesialister kan ha, og skal beskrives ved hjelp av et grensesnitt.

En lege har en beholder som inneholder alle reseptene han eller hun har skrevet ut. Mer om denne beholderen senere i oppgaven.

Personer

Personer har et navn og et unikt nummer. Når en ny person registreres gis personen et nytt løpende (unikt) nummer som starter på null med første personen som opprettes. En person har en beholder over alle personens resepter. Mer om denne beholderen senere i oppgaven.

Noen personer er kvinner, andre personer er menn.

Oppgave 1
Tegn opp klassehierarkiene beskrevet over. Ta også med alle grensesnitt. Du skal ikke ta med data og metoder i klassehierarkiet.

Oppgave 2
Skriv programmene for alle klassene og grensesnittene beskrevet over.

DATASTRUKTUR

I denne oppgaven skal det være fire beholdere som tar vare på hhv. legemidler, resepter, leger og personer. I tillegg har en lege en beholder over alle reseptene vedkommende har skrevet ut, og en person har en beholder som inneholder alle personenes resepter. Klassene som beskriver disse beholderne defineres lenger nede i oppgaven. Legemidlene skal lagres i et objekt av klassen Tabell, reseptene skal lagres i et objekt av klassen EnkelReseptListe, legene skal lagres i et objekt av SortertEnkelListe og personene skal lagres i et objekt av klassen Tabell. Beholderen som inneholder en leges resepter skal være av klassen EldsteForstReseptListe, mens beholderen som inneholder en persons resepter skal være av klassen YngsteForstReseptListe.

Oppgave 3. Tegn opp denne datastrukturen (uten å detaljere hvordan de enkelte beholderne er implementert). Tegn noen legemiddel-objekter, noen lege-objekter, noen person-objekter og noen resept-objekter. La det komme klart frem at en resept er med i mange beholdere.

Noen av disse klassene bygger på grensesnitt som først må defineres:

Oppgave 4. Grensesnitt for beholdere

Skriv programmet for det generiske grensesnittet AbstraktTabell. Det skal ikke være noen restriksjoner på hva slags elementer den abstrakte tabellen skal kunne inneholde. AbstraktTabell beskriver en beholder og du skal kunne:
- sette et objekt inn i tabellen på en oppgitt plass (indeks). Metoden returnerer sann eller usann avhengig om operasjonen gikk bra eller ikke.
- finn et objekt basert på en indeks.
- returnere en Iterator over listen.

Skriv programmet for det generiske grensesnittet AbstraktSortertEnkelListe. En slik liste skal bare kunne inneholde elementer som implementerer grensesnittene Comparable (med seg selv) og Lik. En slik liste skal kunne:
- sette inn et nytt element (i sortert rekkefølge, minste først).
- finne et element basert på en nøkkel av typen String
- returnere en Iterator over listen, slik at innholdet kan bli listet opp i sortert rekkefølge, minste først.

Oppgave 5. Klasser for beholdere

a) Generiske klasser

Skriv den generiske klassen Tabell som implementerer AbstraktTabell.

Klassen skal lagre alle elementene i en array, og arrayens lengde skal oppgis som parameter til konstruktøren. På gruppene vil du få hjelp til å lage iteratoren over listen.

Hvis du har lyst og tid: Når du setter noe inn i Tabellen og det ikke er plass, skal du lage en ny array som er lang nok (innenfor rimelighetens grenser), og så kopiere alle elementene over til den nye arrayen.

Skriv den generiske klassen SortertEnkelListe som implementerer AbstraktSortertEnkelListe som en enveisliste.

b) Ikke generiske klasser

Skriv klassen EnkelReseptListe. Klassen EnkelReseptListe skal inneholde en envisliste med en peker til første og en peker til siste element i listen. Klassen skal kunne ta vare på resepter, og en resept må kunne være med i flere objekter av denne klassen. Metodene i klassen skal kunne sette inn en resept og finne en resept basert på reseptnummeret. Hvis resepten som det letes etter ikke finnes i listen, skal det kastes et unntak. Skriv også en iterator over listen.

Skriv subklassene EldsteForstReseptListe og YngsteForstReseptListe. Når du itererer over den første klassen skal du starte med den eldste resepten (den som ble satt inn først) og gå mot mot yngre (de som ble satt inn sist). Når du itererer i den andre klassen, skal du starte med i den yngste enden.

Hint 1: Forskjellen på de to subklassene til klassen EnkelReseptListe er bare metoden som setter inn en resept.

Hint 2: I både SortertEnkelListe og EnkelReseptListe skal du skrive en iterator. Kan du klare å bruke den samme iteratoren i begge klassene?

FULLSTENDIGE PROGRAMMER

Oppgave 6. Lag enhetstester for alle beholderene.

Svaret på denne oppgaven skal ikke leveres inn, men du bør gjøre den for å overbevise deg om at klassene er riktig programmert.

Skriv et fullstendig testprogram for hver av klassene Tabell, SortertEnkelListe, EldsteForstReseptListe og YngsteForstReseptListe. Prøv å lage programmene slik at både vanlige tilfeller og de fleste spesialtilfellene blir testet.

Oppgave 7. Skriv et ordrestyrt program for leger og resepter.

Nå må du se nøye på datastrukturen du tegnet i oppgave 3.

Det ordrestyrte programmet skal kunne:

Opprette og legge inne et nytt legemiddel.
Opprette og legge inn en ny lege.
Opprette og legge inn en ny person.
Opprette og legge inn en ny resept.

Hente legemiddelet på en resept basert på nummeret til personen som skal ha resepten og reseptens nummer. Siden vi i denne oppgaven ikke har noe data om mengden av legemiddel på lager, betyr dette at vi bare teller ned antallet ganger resepten kan brukes (reit). Om antallet blir null, betyr dette at resepten er ugyldig. Prisen som skal betales skrives ut. Skriv også ut legens navn, personens navn og all dataene du har om legemiddelet på resepten (inkludert antall piller i en eske, hvor stort volum en tube har eller hvor mye virkemiddel det er i en dose).

Lese hele datastrukturen fra en fil som har et gitt filformat. Dette filformatet blir beskrevet i et eget dokument som blir tilgjengelig senest 24. februar her.

Programmet skal kunne håndtere mange former for oppslag og statistikk.
Bruk for-each-løkker for å gå gjennom beholderne.
I programmet du skriver skal du gjøre følgende:

Skriv ut de data som er tilgjengelig om alle legemidler, leger og personer. Legene skal skrives ut i sortert rekkefølge (på navn). Data om resepter skal ikke skrives ut.

-For en gitt person (med et gitt personnummer):
Gå gjennom alle reseptene og skriv ut hvor mange gyldige blå resepter denne personen har og for disse gyldige blå reseptene hvor mange injeksjonsdoser det er igjen.

- Gå gjennom alle legene og for hver eneste lege med avtale skriv ut legens navn og gå gjennom alle legens resepter og skriv ut hvor mange resepter denne legen har skrevet ut på narkotiske legemidler.

- Gå gjennom alle personene og for hver eneste person skriv ut personens navn og gå gjennom alle personens resepter og skriv ut hvor mange gyldige resepter som er skrevet ut på vanedannende legemidler. Skriv helt til slutt ut hvor mange slike gyldige resepter det er totalt og hvor mange av disse er til kvinner og hvor mange er til menn.

Hvis du har tid og lyst (å skrive ut statistikk blir mye morsommere hvis det er mye data): Skrive hele datastrukturen til fil på det samme formatet som du brukte da du leste inn fra fil. Rekkefølgen på reseptene behøver ikke opprettholdes ved skriving/lesing til/fra fil.

Slutt obligatorisk oppgave 4.

## B: Origin of the evaluation criteria

Criteria number 10, 11, 16, 17 and 19 from table 3.3 are taken directly from the assignment description, while criteria 12, 13, 14, 15, 18 and 20 are our interpretation of the text. Still, the assignment description does not state directly that the doctor and person is supposed to be classes, but they are listed under the section called “The class hierarchy” and under subsections called “Doctors” and “Persons”.

Criteria number 21, 22, 23, 24, 25, 27, 28 and 29 from table 3.5 are taken directly from the assignment description. Criteria 26 is our interpretation, mainly based on discussions with the teaching assistants writing the example solution for the assignment.

Criteria number 30, 31, 32, 34, 35 and 36 from table 3.6 are taken directly from the assignment description. Criteria number 40, 41, 42 and 43 from table 3.7 are our interpretation of the assignment text. The remaining criteria, 33, 37, 38, 39, 44, 45 and 46 we have written our selves based on what we think is good solutions and that will give us relevant information about the comprehension.

## C: Details form the analysis

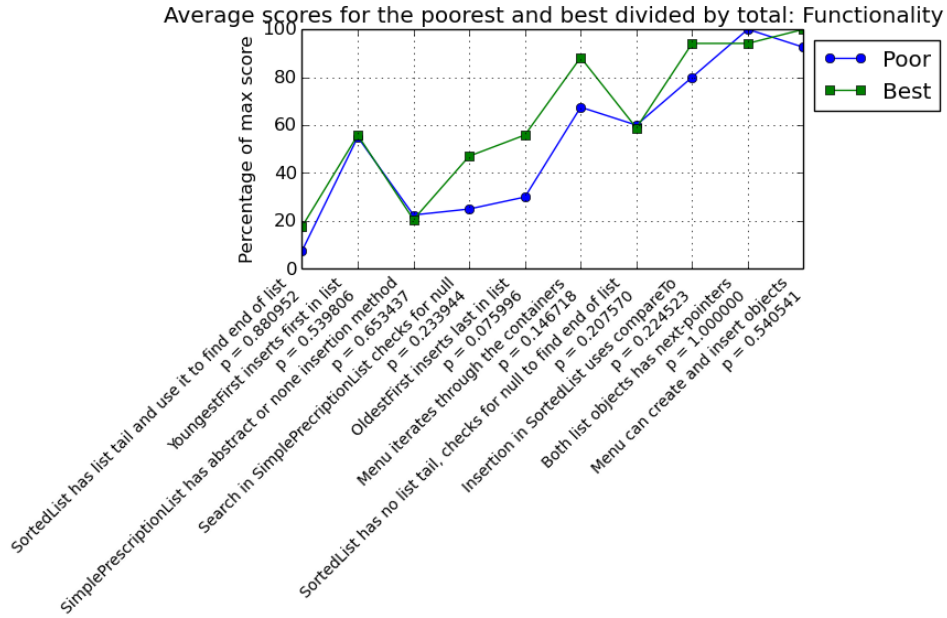


Figure 6.1: Average scores for all students functionality, students divided by total score.

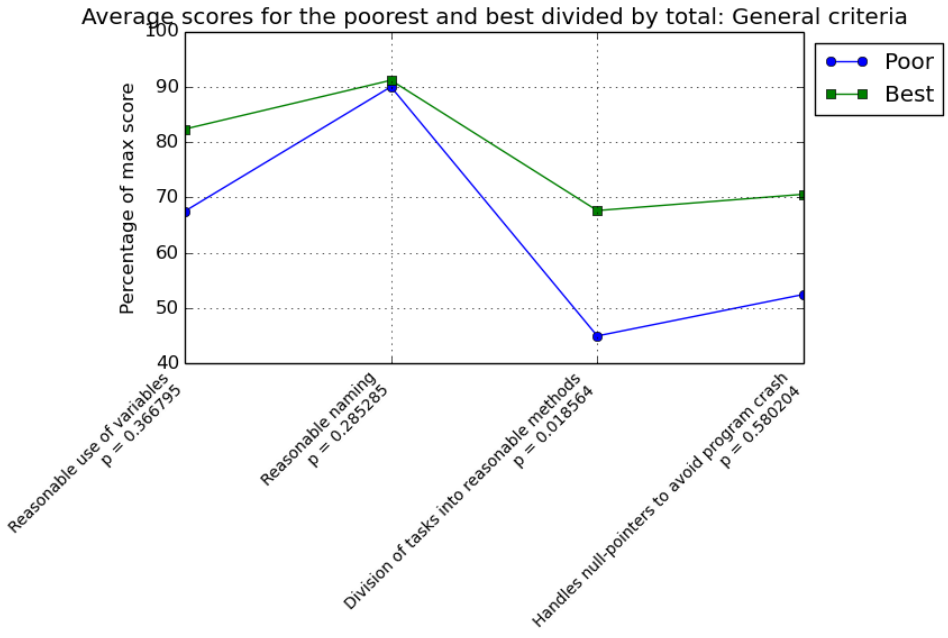


Figure 6.2: Average scores for all students general criteria, students grouped by total score.

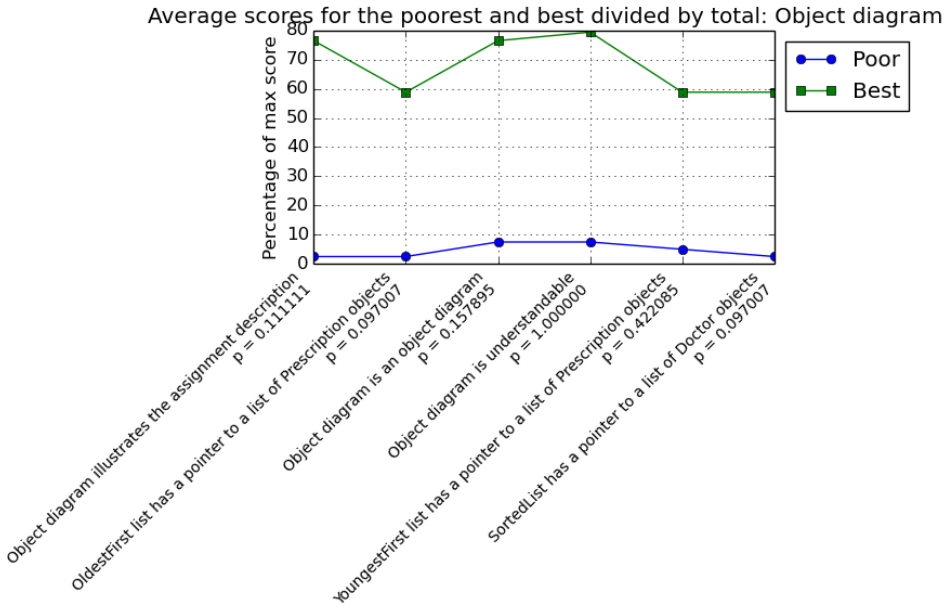


Figure 6.3: Average scores for all students object diagram, students divided by total score.

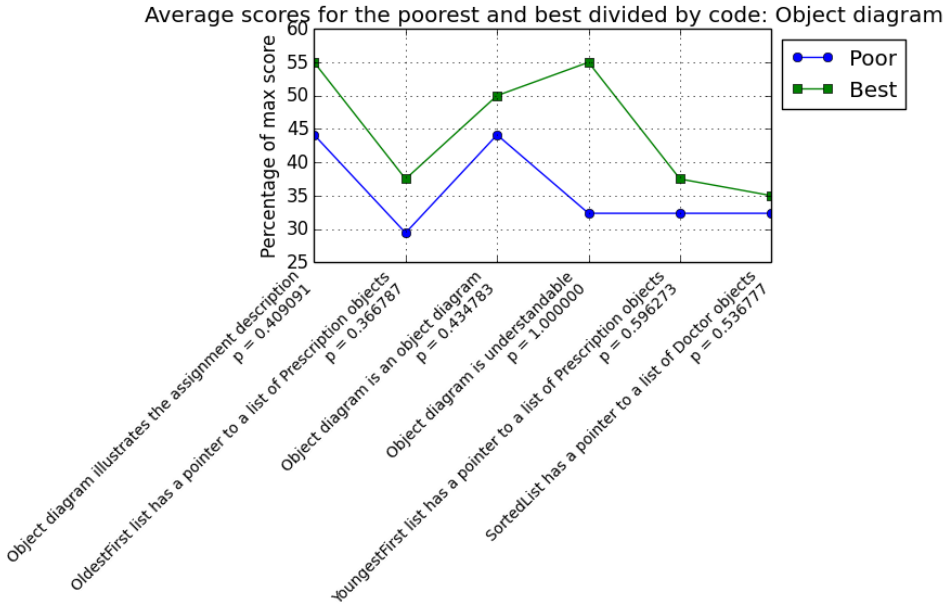


Figure 6.4: Average scores for the object diagram, students grouped by code score.

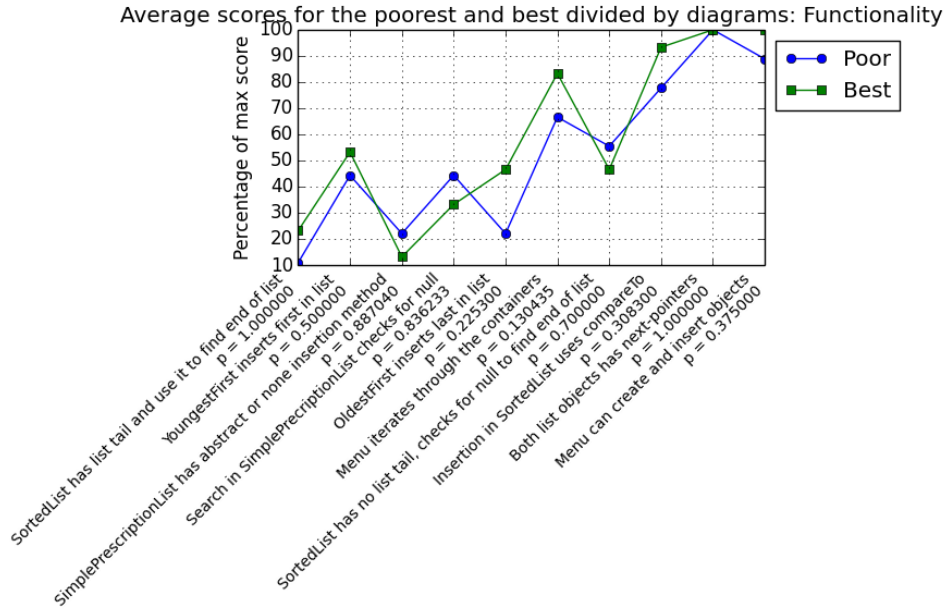


Figure 6.5: Average scores for the functionality, students grouped by diagram score.

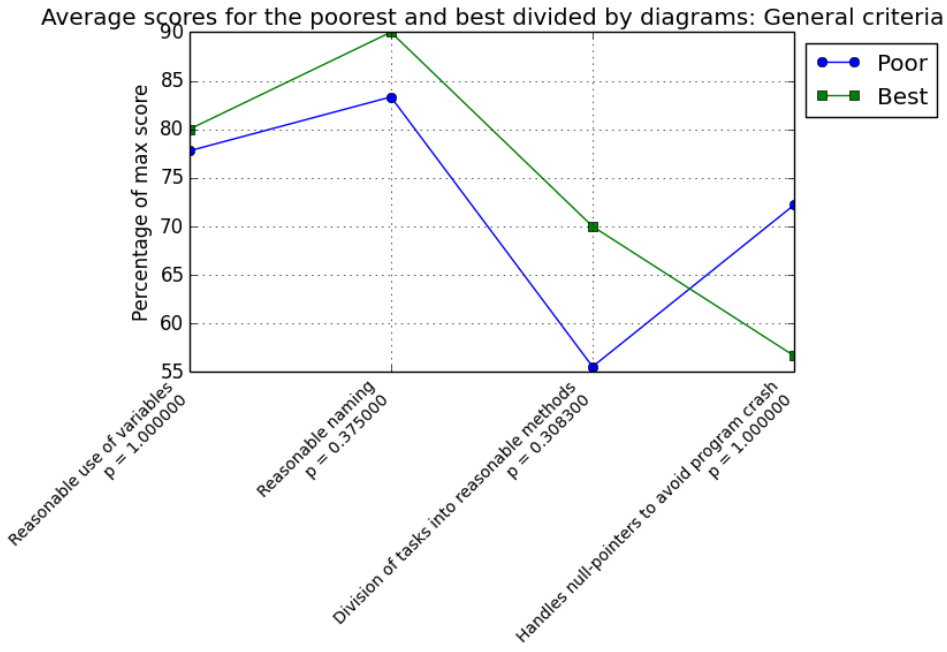


Figure 6.6: Average scores for the general criteria, students grouped by diagram score.